Limited Warranty Information

I. CUSTOMER OBLIGATION

A. CUSTOMER assumes full responsibility that this computer hardware, (the "Equipment") and/or software (the "Software") meets the specifications, capacity, capabilities, versatility, and other requirements of CUSTOMER.

B. CUSTOMER assumes full responsibility for the condition and effectiveness of the operating environment in which the Equipment and Software is to function, and for its installation.

II. RADIO SHACK LIMITED WARRANTIES AND CONDITIONS OF SALE

A. For a period of ninety (90) calendar days from the date of the Radio Shack sales ticket, RADIO SHACK warrants to the original CUSTOMER that the Equipment and the cassettes and/or diskettes containing software programs are free from defects. This warranty is only applicable to purchases from RADIO SHACK company-owned Computer Centers, retail stores and through RADIO SHACK franchisees and dealers. The warranty is void if the unit's case or cabinet has been opened, or if the unit has been subjected to improper or abnormal use. If a defect occurs during the warranty period, the defective Equipment must be returned to a Radio Shack Computer Center, a Radio Shack retail store, participating franchisee or dealer for repair, along with a copy of the sales ticket or lease agreement. The original CUSTOMER'S sole and exclusive remedy in the event of a defect is limited to the correction of the defect by repair, replacement, or complete refund, at RADIO SHACK'S election and sole expense. RADIO SHACK has no obligation to replace or repair expendable items.

B. RADIO SHACK makes no warranty as to the design, capability, capacity, or suitability for use of the Equipment or Software. Software is licensed on an "AS IS" basis, without warranty. CUSTOMER'S exclusive remedy, in the event of a software defect is its repair or replacement within thirty (30) calendar days of the date of purchase upon its return to a Radio Shack Computer Center, Radio Shack retail store, participating franchisee or dealer along with the sales ticket.

C. Except as provided herein no employee, agent, franchisee dealer or other person is authorized to give any warranties of any nature on behalf of RADIO SHACK.

D. Except as provided herein, <u>RADIO SHACK MAKES NO WARRANTIES, INCLUDING WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.</u>

F. Some states do not allow limitations on how long an implied warranty lasts, so the above limitation(s) may not apply to CUSTOMER.

III. LIMITATION OF LIABILITY

A. EXCEPT AS PROVIDED HEREIN, RADIO SHACK SHALL HAVE NO LIABILITY OR RESPONSIBILITY TO CUSTOMER OR ANY OTHER PERSON OR ENTITY WITH RESPECT TO ANY LIABILITY, LOSS OR DAMAGE CAUSED OR ALLEGED BE CAUSED DIRECTLY OR INDIRECTLY BY "EQUIPMENT" OR "SOFTWARE" SOLD, LEASED, LICENSED OR FURNISHED BY RADIO SHACK, INCLUDING, BUT NOT LIMITED TO, ANY INTERRUPTION OF SERVICE, LOSS OF BUSINESS OR ANTICIPATORY PROFITS OR CONSEQUENTIAL DAMAGES RESULTING FROM THE USE OR OPERATION OF THE "EQUIPMENT" OR "SOFTWARE." IN NO EVENT SHALL RADIO SHACK BE LIABLE FOR LOSS OF PROFITS, OR ANY INDIRECT, SPECIAL, OR CONSEQUENTIAL DAMAGES ARISING OUT OF ANY BREACH OF THIS WARRANTY OR IN ANY MANNER ARISING OUT OF OR CONNECTED WITH THE SALE, LEASE, USE OR ANTICIPATED USE OF THE "EQUIPMENT" OR SOFTWARE".

NOTWITHSTANDING THE ABOVE LIMITATIONS AND WARRANTIES, RADIO SHACK'S LIABILITY HEREUNDER FOR DAMAGES INCURRED BY CUSTOMER OR OTHERS SHALL NOT EXCEED THE AMOUNT PAID BY CUSTOMER FOR THE PARTICULAR "EQUIPMENT" OR "SOFTWARE" INVOLVED.

B. RADIO SHACK shall not be liable for any damages caused by delay in delivering or furnishing any Equipment or Software.

C. No action arising our of any claimed breach of this WARRANTY or transactions under this WARRANTY may be brought more than two (2) years after the cause of action has accrued or more than four (4) years after the date of the Radio Shack sales ticket for the Equipment or Software whichever first occurs.

D. Some states do not allow the exclusion or limitation of incidental or consequential damages, so the above limitation(s) or exclusion(s) may not apply to CUSTOMER.

IV. RADIO SHACK SOFTWARE LICENSE

RADIO SHACK grants to CUSTOMER A non-exclusive, paid-up license to use the RADIO SHACK application or system Software and/or the RADIO SHACK system Software (including firmware) installed in or provided with the Equipment on <u>one</u> computer, subject to the following provisions:

A. Except as otherwise provided in this Software License, applicable copyright laws shall apply to the Software.

B. Title to the medium on which the Software is recorded (cassette and/or diskette) or stored (ROM) is transferred to CUSTOMER, but not title to the Software.

C. CUSTOMER shall not use, make, manufacture, or reproduce copies of Software except for use on <u>one</u> computer and as is specifically provided in this Software License.

D. CUSTOMER is permitted to make additional copies of the Software <u>only</u> for backup or archival purposes or if additional copies are required in the operation of <u>one</u> computer with the Software, but only to the extent the Software allows a backup copy to be made.

E. CUSTOMER may resell or distribute unmodified copies of the Software provided CUSTOMER has purchased one copy of the Software for each one sold or distributed. The provisions of this Software License shall also be applicable to third parties receiving copies of the Software from CUSTOMER.

F. All copyright notices shall be retained on all copies of the Software.

V. APPLICABILITY OF WARRANTY

A. The terms and conditions of this WARRANTY are applicable between RADIO SHACK and CUSTOMER to either a sale of the Equipment and/or Software License to CUSTOMER or to a transaction whereby RADIO SHACK sells or conveys such Equipment and/or Software to a third party for lease to CUSTOMER.

B. The limitations of liability and warranty provisions herein shall insure to the benefit of RADIO SHACK, the owner and/or licensor of RADIO SHACK Software to RADIO SHACK, and any author or manufacturer of computer hardware or Equipment sold or Software licensed by RADIO SHACK.

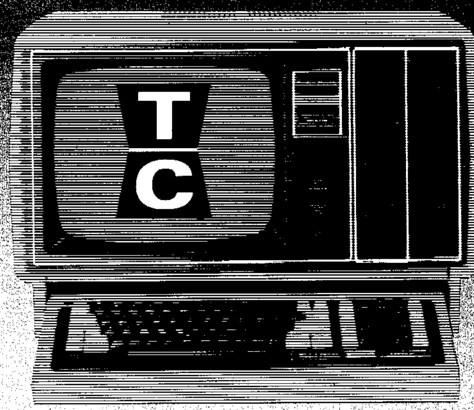
VII. STATE LAW RIGHTS

The warranties granted herein give the <u>original</u> CUSTOMER specific legal rights, and the <u>original</u> CUSTOMER may have other rights which vary from state to state.

Radio Shack

TRS-80 MODEL 16

ASSEMBLER-16 PROGRAMMING PACKAGE



CUSTOM MANUFACTURED IN U.S.A. BY RADIO SHACK, A DIVISION OF TANDY CORPORATION

ASSEMBLER-16

PROGRAMMING PACKAGE

TO OUR CUSTOMERS...

The Assembler-16 Programming Package contains three systems for developing programs in MC68ØØØ object code:

> **The EDITOR (EDIT16)** which allows you to create and edit "source" assembly language programs
>
> **The ASSEMBLER-16 (ASM16)** which assembles the source program into an intermediate 68ØØØ object code program.
>
> **The LINKER (LINK16)** which links the intermediate object code program into an absolute executable program file.

Also, as part of the TRSDOS-16, you can use:

> **The DEBUGGER (DEBUG)** for debugging the absolute program.

ABOUT THIS MANUAL

This manual contains two sections:

> **Section 1/ Operations** explains how to use the four systems.
>
> **Section 2/ Assembler-16 Reference Guide** references the assembly language required by the Assembler-16.

The terms and notations the manual uses are:

ALL CAPS
indicates what will appear on your screen or what you should type.

<KEYBOARD CHARACTER>
indicates keys you press.

<u>lowercase underlined</u>
represents words, characters, or values to be supplied by you or the system.

<u>filespec</u>
is a standard TRSDOS-16 file specification, described in the TRSDOS-16 Manual, having the general form:
    <u>filename</u>/<u>ext</u>.<u>password</u>:<u>drive</u>(<u>disk name</u>)

The notations and terms accepted by the Assembler-16 are in the beginning of Section 2.

TABLE OF CONTENTS

CHAPTER 1

SAMPLE SESSION

CHAPTER 1/ SAMPLE SESSION

This chapter shows how to use the Assembler-16 Programming Package to create, debug, and execute a 68ØØØ object program.

This is for demonstration purposes only. For complete information on each system's commands, listings, and error messages, see the appropriate chapter.

DEVELOPING A 68ØØØ PROGRAM

To develop a 68ØØØ program file, follow these steps:

1. Create one or more source program files (with the Editor)

2. Assemble the source files into intermediate object files (with the Assembler)

3. Create a linker control file (with the Editor)

4. Using the control file, link the intermediate files into an absolute program file (with the Linker)

1. Creating a Source File

To create the source file, type:

    EDIT16 <ENTER>

which loads the Editor. At the C? prompt type:

    IN <ENTER>

which enters the insert mode. At the I? prompt, insert this program:

```
BEGIN      LDA     .AØ,SVC BLOCK     *load svc block
           MOVW    @AØ,#8            *store vdchar svc
           MOVW    6@AØ,#65          *store 'A
           BRK     #Ø                *execute vdchar
```

```
                LDA          .AØ,SVC BLOCK        *load svc block
                MOVW         @AØ,#264             *store jp2dos svc
                BRK          #Ø                   *execute jp2dos svc
SVC BLOCK
                RDATAB       32,Ø                 *reserve svc block
                END          BEGIN
```

Use <ENTER> to enter each line; <TAB> to tab between columns. (The Editor displays the <TAB> as an $\pm$ character rather than tabbing.)

This assembly language program contains Assembler-16 instructions, described in Section II, and TRSDOS-16 SVCs, described in the TRSDOS-16 Operating System Manual.

If you need to edit the program, see Chapter 2. Otherwise, save it and exit the Editor with:

```
! <ENTER>
SAVE SAMPLE/SRC <ENTER>
QUIT <ENTER>
```

You should now have a source disk file named SAMPLE/SRC.

> Note: After using an SVC, you should normally check offsets 2 and 3 for an error code. For simplicity, this program does not do this.

2. Assembling an Intermediate File

To assemble SAMPLE/SRC, type:

```
ASM16 SAMPLE <ENTER>
```

which causes the Assembler to load and then assemble SAMPLE/SRC into "intermediate", relocatable object code. It then saves the intermediate code on disk as a file named SAMPLE/OBJ.

3. Creating a Control File

To create a linker control file, load the Editor and insert this program:

```
INCLUDE SAMPLE
```

END

INCLUDE and END are directives controlling the Linker (discussed in Chapter 4).

SAVE this program as a file named SAMPLE/CTL and exit the Editor.

4. Linking an Absolute Program File

To link the program file, type (at TRSDOS-16 Ready):

```
LINK16 SAMPLE <ENTER>
```

which loads the Linker and then loads SAMPLE/CTL.

The Linker links the one file which SAMPLE/CTL directs it to INCLUDE -- SAMPLE/OBJ -- to absolute addresses beginning with address ØØØØ. (You could INCLUDE other intermediate files, as well.)

The Linker saves this as an executable program file named SAMPLE.

Refer to Chapter 4 for a complete listing of all the options to the linker command.

EXECUTING THE PROGRAM

Since SAMPLE is an absolute, executable program file, you can execute it from the TRSDOS-16 Ready mode. At TRSDOS-16 Ready, type:

```
SAMPLE <ENTER>
```

TRSDOS-16 loads and executes SAMPLE beginning at the "relative" address of ØØØØ.

Note that address ØØØØ is relative. TRSDOS-16 loads itself and the Debugger, if present, in an area of memory that is "invisible" to the user. The relative address of ØØØØ is actually the first address available after TRSDOS-16 and the Debugger.

Because TRSDOS-16 uses relative addresses, you need not be concerned about loading your program over system memory.

DEBUGGING THE PROGRAM

If you need to debug the program, you can use the Debugger. At TRSDOS-16 Ready, type:

```
DEBUG ON <ENTER>
SAMPLE <ENTER>
```

which turns on the Debugger and then loads SAMPLE. If the Debugger does not activate, you will need to configure it into system memory. Appendix D explains how.

Once SAMPLE is loaded into the Debugger, you can use any of the Debugger commands. For example:

```
N <ENTER>
```

executes the SAMPLE's first instruction.

```
V 1A <ENTER>
```

displays the contents of addresses ØØ1A through ØØ29.

To exit the Debugger and return to TRSDOS-16 Ready, type:

```
O <ENTER>
```

CHAPTER 2

THE EDITOR

CHAPTER 1/ THE EDITOR

The Editor is a set of commands that allows you to create and edit text files.

You can use the Editor with:

1. The Assembler-16
2. The COBOL Compiler
3. The TRSDOS-16 DO command and Configuration command file

It allows you to:

1. Create files.

You can write your own programs and save them to disk for future use.

2. Edit existing files.

You can change the program lines or contents of a file.

3. Combine files.

You can combine multiple programs together into one program.

LOADING THE EDITOR

This command, typed at TRSDOS-16 Ready, loads the Editor:

EDIT16 source filespec {options}

source filespec is optional. It causes the Editor to CONCATenate the specified source filespec.

The options are:

W=drive tells the Editor which drive to use as its work file.

M=drive tells the Editor which drive to use as a "scratch file" during a MOVE.

WORK AND SCRATCH FILES

W=drive
As the Editor creates or edits a program, it does not use its own memory to do so. It stores your program on disk in a temporary "work file". You must have enough space on disk for this file, otherwise you get a disk-full error.

The W=drive option tells the Editor which diskette to use for the work file. In this way you can save your good diskettes from excessive writes and deletions.

If drive is not a valid drive specification, EDIT16 will not load and you will be returned to TRSDOS-16 Ready.

M=drive
When you use the Editor's MOVE command, the Editor creates another temporary disk file called a "scratch file".

The M=drive option tells the Editor which diskette to use as its "scratch file". Again, this will save wear and tear on your good diskettes and files.

If drive is not a valid drive specification, the Editor will use the first available drive for the scratch file.

SAMPLE SESSION

To enter the Editor, type:

```
EDIT16 <ENTER>
```

and the Editor displays the prompt:

```
C?................................
```

This is the Editor's Command mode. You can use any of the Editor's commands in this mode.

To create a program in the Editor, you must get in the Insert mode. At the C? prompt, type:

```
IN <ENTER>
```

The Editor displays the I? prompt -- indicating the Insert mode. Type in the following program lines. The asterisk (*) indicates comment lines:

```
* THIS IS A PROGRAM <ENTER>
* THAT WILL DEMONSTRATE <ENTER>
* ALL OF THE TRSDOS-16 <ENTER>
* EDITOR'S COMMANDS <ENTER>
<ENTER>
```

When you press <ENTER>, the Editor exits the Insert mode. The Editor keeps this file in its work file until you delete the lines, SAVE the file, or exit the Editor.

To see what you've just entered in the Editor's work file, type:

```
LIST ALL <ENTER>
```

and the Editor returns a complete program listing:

```
* THIS IS A PROGRAM
* THAT WILL DEMONSTRATE
* ALL OF THE TRSDOS-16
* EDITOR'S COMMANDS
```

Notice there are no line numbers in this program. To give it line numbers, you must first save your program by typing:

```
SA SAMPLE1/PRO <ENTER>
```

This writes the program as SAMPLE1/PRO to disk.

After saving the program, delete all information from the Editor's work file by typing:

```
DE ALL <ENTER>
```

(The Editor prompts you with CANCEL = 'X'; type <ENTER> to continue the DElete command. See DELETE for details.)

Next type:

```
CO SAMPLE1/PRO <ENTER>
```

to load -- CONCATenate -- the program into the Editor's work file. (See the appropriate Editor's command for details on its use.)

Your program is now in the Editor's work file with numbers:

```
1 * THIS IS A PROGRAM
2 * THAT WILL DEMONSTRATE
3 * ALL OF THE TRSDOS-16
4 * EDITOR'S COMMANDS
```

An alternate way to load your program into the work file (after you've SAVEd it) is to exit the Editor. Type:

```
QU <ENTER>
```

Then reload the Editor and the file at the same time by typing:

```
EDIT16 SAMPLE1/PRO <ENTER>
```

LINE NUMBERING

The Editor provides for two types of line numbrs:

- relative line numbers
- absolute line numbers

Relative line numbers

When you initially create a program, the Editor does not assign line numbers. You can reference these unnumbered lines via relative line numbers.

Relative line numbers are:

$	the current line
$-<u>n</u>	the current line minus <u>n</u> lines
$+<u>n</u>	the current line plus <u>n</u> lines

For example, to refer to a line five lines before the current line, use the relative line number $-5. To refer to a line nine lines after the current line, use $+9.

Absolute line numbers

An absolute line number is the actual number which the Editor gives a line when you first CONCATenate the file into a work file.

An "absolute line number" can be:

- a whole number, including zero (Ø)
- START
- END

Whenever the Editor concatenates a file into an empty work file, it assigns each line a whole number beginning with 1.

Although the Editor never assigns a line Ø, you can use it to insert lines at the beginning of the program. The commands CONCATenate, MOVE, and INSERT are the only ones that recognize a reference to line Ø.

START and END refer specifically to the first or last line of the program.

REFERENCING PROGRAM LINES

The following commands allow you to specify one <u>line</u> or a group of <u>lines</u>.

<u>line</u>
refers to a single line of the program. This single line can be either an "absolute line number" or a "relative line number".

Some examples of <u>line</u> as an absolute line number are:

START	Refers to first line of program
23	Refers to line number 23
1Ø58	Refers to line number 1Ø58
END	Refers to last line of program
Ø	Refers to the line preceding the first line of the program

Some examples of <u>line</u> as a relative line number are:

$	Refers to the last line the work file

	displayed -- the current line
$-7	Refers to the line that is seven lines before current line.
$+6	Refers to line that is six lines after the current line.

<u>lines</u>

When you see <u>lines</u> in the command's syntax, you can enter:

- One line

to indicate a single line only.

- ALL lines

to indicate every line of the work file.

- A pair of lines

to indicate all lines between and including the pair of lines. To separate the line pair, use a comma (,), period (.), or hyphen (-).

When you specify a pair of lines, you cannot mix absolute and relative line numbers. For example:

    2-4 and $-2,$

are valid pairs of lines to reference. Both 2 and 4 are absolute line numbers. Both $-2 and $ are relative line numbers.

    $,2 and START-$

are not valid because $ and 2 and START and $ reference both relative and absolute line numbers within one pair.

When specifying a pair of lines in a command, be sure to put the earliest line in the program first. You can use the pair $-6,$-4, but not $-4,$-6.

Some more examples of pairs of lines are:

START,23	Refers to all lines from the first to line 23.
971-1Ø23	Refers to lines 971 through 1Ø23.

ALL	Refers to the entire contents of the work file.
$-7,$-2	Refers to the lines from the seventh before the current line to the second before the current line.
146,END	Refers to lines 146 to the end.
$.$+3	Refers to the lines from the current line to the line three lines after the current line.
START.END	Refers to the entire contents of the work file.

SPECIFYING STRINGS

Three more terms to know when specifying a string to search for are:

<u>delimiter</u>	indicates a character used to mark the beginning and end of distinct variables.
<u>string</u>	indicates a set of characters. These can be any alphanumeric characters including blanks.
G	indicates a Global search. When G is specified, the Editor will search for each occurrence of the specified string within the given range. Without G, the Editor stops at the first occurrence of string.

USING THE <BREAK> KEY

You can use the <BREAK> key in the Editor with these commands:

CONCAT	SAVE
LIST	SEARCH
MOVE	STRING
PRINT	

When you press <BREAK> with any of these commands, the Editor will display the last line of operation and will return to the command mode -- C? prompt.

NOTE: Be careful when using <BREAK> with the SAVE and STRING commands. If you press <BREAK> while executing one

of these commands, part of the work file (with STRING) or the disk file (with SAVE) will be altered.

ENTERING AN EDITOR COMMAND

The following pages list all the Editor's commands. You may enter them by typing either:

- the entire command (SAVE FILE <ENTER>)
- the first two letters of the command (SA FILE <ENTER)

Most commands allow you to specify an expression.

If the expression begins with an alpha-character (A through Z), you must leave at least one blank space between the command and the expression (SA NEWFILE rather than SANEWFILE).

If the expression begins with a number, you do not need to type an intervening space (both IN 1ØØ and IN1ØØ are correct).

CHANGE

    CH line

allows you to change line.

line is optional; if omitted, the Editor displays the current line for you to change.

Once you enter this command, the Editor displays the I? prompt. To change the line, type the new line followed by <ENTER>.

If you decide not to change the line, press <ENTER>.

Examples

With our sample program inserted, type (at the C? prompt):

```
CH 1 <ENTER>
```

and the Editor displays:

```
1 * THIS IS A PROGRAM
I?
```

Change line 1 by typing:

```
* THIS IS A NEW PROGRAM <ENTER>
```

The Editor then displays the new line 1 and returns to the Editor's command mode.

CONCAT

CO (line) filespec (lines)

inserts (CONCATenates) the contents of filespec into the Editor.

line is optional and can only be used when the work file already contains a program. It indicates where in the work file to insert filespec. If omitted, filespec is inserted at the current position.

lines is optional and tells the Editor to only CONCATenate the specified lines from filespec. If omitted, the entire file is inserted.

Both line and lines must be enclosed in parentheses ( ).

Examples

Before entering this example, delete everything in the work file by typing:

DE ALL <ENTER>

To CONCATenate SAMPLE1/PRO into the work file, type:

CO SAMPLE1/PRO <ENTER>

The Editor loads the file SAMPLE1/PRO from disk, displaying the last line with its new line number.

To see the entire listing of the file, type LI ALL <ENTER> and the Editor displays:

```
1 * THIS IS A PROGRAM
2 * THAT WILL DEMONSTRATE
3 * ALL OF THE TRSDOS-16
4 * EDITOR'S COMMANDS
```

To CONCAT lines into a program already in the work file, type:

CO (2) SAMPLE1/PRO (2-3) <ENTER>

The Editor inserts lines two and three of SAMPLE1/PRO after the second line of the current program in the Editor.

(After the Editor CONCATenates lines, it displays the last of the lines being inserted.)

Type LI ALL <ENTER> to see all of the Editor's contents:

```
1 * THIS IS A PROGRAM
2 * THAT WILL DEMONSTRATE
  * THAT WILL DEMONSTRATE
  * ALL OF THE TRSDOS-16
3 * ALL OF THE TRSDOS-16
4 * EDITOR'S COMMANDS
```

When the Editor concatenates lines of a file into a program already in a work file, it doesn't add line numbers to the most recently CONCATenated lines.

DELETE

DE lines

deletes lines.

If you do not specify lines, the Editor deletes the current line.

If you specify lines as ALL, the Editor prompts you with:

CANCEL = 'X'

Type X <ENTER> to cancel the DElete command. If you really do want to delete all the lines, press <ENTER>.

Examples

With the Editor's current position at the END of the work file, you can delete the two lines we inserted in the last example by typing:

DE $-3,$-2 <ENTER>

The program will again be:

```
1 * THIS IS A PROGRAM
2 * THAT WILL DEMONSTRATE
3 * ALL OF THE TRSDOS-16
4 * EDITOR'S COMMANDS
```

To delete the entire program, type:

DE ALL <ENTER>

Before actually deleting the lines, the Editor prompts:

CANCEL = 'X'

Type X <ENTER> to cancel the DElete ALL command. (If you really do want to delete all the lines, press <ENTER>).

INSERT

IN line

enters the Insert mode, displays the I? prompt, and allows you to insert lines after the referenced line.

To terminate the Insert mode, type ! <ENTER> or simply <ENTER> at the beginning of an Insert line. The Editor will display the last line that you inserted, followed by the command mode prompt -- C?.

If you don't specify line, insertion begins after the current line.

Examples

If you have the sample program entered in the Editor and want to insert new lines after the fourth line, type:

```
IN 4 <ENTER>
```

The Editor displays:

```
4 * EDITOR'S COMMANDS
I?.........................
```

You are now in the Insert mode and can now insert lines after line 4. Type:

```
BEGIN<TAB>LD<TAB>.A0,#TABLE<TAB>*load start of table <ENTER>
<TAB>BNE<TAB>DONE<TAB>*if no match go to DONE <ENTER>
<ENTER>
```

When you press the <TAB> key in the Insert mode, you'll see +_. This represents the <TAB> key. The tabstops are preset to multiples of eight, i.e., tabstops at 8, 16, 24, 32, etc. (To set your own tabs, see the TAB command later in this chapter.)

To see the tabbed inserted lines and the rest of the program, at the C? prompt, type:

```
LI ALL <ENTER>
```

and the Editor displays:

```
1 * THIS IS A PROGRAM
2 * THAT WILL DEMONSTRATE
3 * ALL OF THE TRSDOS-16
4 * EDITOR'S COMMANDS
  BEGIN   LD     .AØ,#TABLE        *load start of table
          BNE    DONE              *if no match go to DONE
```

To insert lines at the beginning of a program, type:

INØ <ENTER>

you'll see the I? prompt. You can now insert lines which will precede the first line of the program. For example, type:

* LET'S LOOK AT THE EDITOR <ENTER>
<ENTER>

This inserts * LET'S LOOK AT THE EDITOR as the first line of the sample program.

To insert a line at the end of the program, type:

IN END <ENTER>

You can now add lines at the end of the program. Type:

<TAB>MOV<TAB>.D2,.DØ+<TAB>*otherwise move element number <ENTER>
<ENTER>

to enter one more line.

Save this new program by typing:

SAVE SAMPLE2/PRO <ENTER>

LIST

LI lines

displays lines and positions the Editor to the last line listed.

lines is optional; if omitted, the Editor lists the current line.

If you attempt to list more than 21 lines (more than the screen can display at one time), the Editor displays the first 21 lines and then returns the message:

```
CANCEL = 'X'
```

If you press <ENTER>, the display will continue. If you type X <ENTER>, the Editor stops the LISTing and returns to the command prompt.

Examples

```
LI ALL <ENTER>
```

returns a complete listing of the current program. In this case, the Editor displays:

```
  * LET'S LOOK AT THE EDITOR
1 * THIS IS A PROGRAM
2 * THAT WILL DEMONSTRATE
3 * ALL OF THE TRSDOS-16
4 * EDITOR'S COMMANDS
  BEGIN   LD      .AØ,#TABLE      *load start of table
          BNE     DONE            *if no match go to DONE
          MOVE    .D2,.DØ         *otherwise move element number
```

To obtain a listing of the first unnumbered line at the beginning of the program, type:

```
LIST START <ENTER>
```

and the Editor returns this listing:

```
  * LET'S LOOK AT THE EDITOR
```

MOVE

MO (<u>lines</u>) TO <u>line</u>

duplicates <u>lines</u> and places them after <u>line</u>.

<u>lines</u> is optional. If omitted, the Editor moves the current line.

<u>line</u> is optional. If omitted, the Editor places <u>lines</u> after the current line.

You must enclose <u>lines</u> with parentheses (). If you do not, the Editor returns an error.

Note: Use the DELETE command (See DELETE) if you want the MOVEd line(s) deleted from the original position in the file.

Examples

```
MO (START)TO END
```

moves the first line of the program to the end of the program.

```
MO (1-2)TO$
```

moves all lines between and including lines 1 and 2 to follow the current line.

```
MO($-2)TO $
```

moves the line two places before the current line to follow the current line.

```
MO TO 4
```

moves current line to follow line 4.

```
MO
```

moves current line to follow current line. (Has the effect of repeating the current line).

Use the DElete command to get rid of any of the lines you MOved that you don't want duplicated in your program. First LI ALL lines, then type:

```
DE$-3,$ <ENTER>
DE$-3 <ENTER>
DE$-3 <ENTER>
```

This returns the program to the contents we will reference in the remainder of this section.

POSITION

PO + n
PO - n

moves the Editor's current position plus (+) or minus (-) n lines and then displays the new current line.

Typing PO is optional; if you simply type +n or -n, you can position the Editor.

n is optional. If you specify it, n can be any whole number. If n is greater than the number of lines in the program, the Editor returns:

ITEM NOT FOUND

and moves the current position to the beginning or end of the program (beginning if you specified minus (-), end if you specified plus (+)).

If you do not specify n, the Editor uses 1.

Once you enter the POsition command, whenever you simply press <ENTER>, the Editor will move one line forward or backward, depending on the POsition command you originally entered. To exit this command, enter another of the Editor's commands.

Examples

Using the sample program, SAMPLE2/PRO, position the Editor to the end by listing the entire program. Type:

LI ALL <ENTER>

and the Editor displays the current program:

```
  * LET'S LOOK AT THE EDITOR
1 * THIS IS A PROGRAM
2 * THAT WILL DEMONSTRATE
3 * ALL OF THE TRSDOS-16
4 * EDITOR'S COMMANDS
  BEGIN   LD      .AØ,#TABLE      *load start of table
          BNE     DONE     *if no match go to DONE
          MOV     .D2,.DØ *otherwise move element number
```

The Editor's position at this time is the last line of the program. To position the Editor to line 2, type:

```
PO -5 <ENTER>
```

The Editor displays:

```
2 * THAT WILL DEMONSTRATE
C?..............................
```

To position the Editor to line number 1, type:

```
 - <ENTER>
```

and the Editor now displays:

```
1 * THIS IS A PROGRAM
C?..............................
```

PRINT

PR lines

prints lines on the printer.

lines is optional. If you omit lines, the Editor prints only the current line.

If lines include absolute line numbers, the Editor prints them also.

Before printing, PRINT moves the paper to the TOP OF FORM.

Examples

PR 2 <ENTER>

prints line 2 on the printer.

PR ALL <ENTER>

prints the entire listing on the printer.

PR <ENTER>

prints the current line on the printer.

QUIT

QU

terminates the Editor and returns to TRSDOS-16.

If you attempt to QUit the Editor without SAVEing the current work file, you'll see the prompt:

```
**NO FILES SAVED**
**QUIT EDITOR? Y OR N**
```

This warns you that you haven't saved the file you were working on. If you don't want to SAVE it, type:

Y <ENTER>

The Editor then terminates and returns to TRSDOS-16 Ready.

If you do not want to exit the Editor without SAVEing your program, type:

N <ENTER>

You will now see the last line, followed by the Editor's command prompt.

Examples

QU <ENTER>

(if you have saved your file) terminates the Editor and displays:

```
**EDITOR TERMINATED**

TRSDOS-16 Ready....................................
```

RELABEL

RE <u>lines</u>

Sequentially reorders the local label numbers between two global label definition lines. (See the Assembler Reference Section for details on local and global labels.)

If either of the lines you indicate does not contain a global label definition, the Assembler will use the next line which does contain one.

When the Editor alters <u>lines</u>, it displays all altered lines followed by the last altered line. That is, it repeats the last line.

Use this command only on source text that uses the local label format.

Examples

Type in the example program using local and global labels. (Be sure to first delete all lines currently in the Editor.)

```
$1<TAB>DATAL<TAB>1Ø <ENTER>
GLOBAL ONE <ENTER>
$7<TAB>BE<TAB>$5 <ENTER>
$5<TAB>BL<TAB>$7 <ENTER>
GLOBAL TWO <ENTER>
$Ø<TAB>NOP <ENTER>
<TAB>LDA<TAB>.AØ,$1 <ENTER>
GLOBAL THREE <ENTER>
! <ENTER>
```

(After typing in this program, SAVE it, DElete ALL lines in the work file, and then CONCATenate it to have the Editor give it line numbers.) When you LIst it, the program now appears as:

```
1 $1      DATAL   1Ø
2 GLOBAL ONE
3 $7      BE      $5
4 $5      BLT     $7
5 GLOBAL TWO
6 $Ø      NOP
7         LDA     .AØ,$1
8 GLOBAL THREE
```

The Editor would execute the following commands in these ways:

RE1,8 <ENTER>

causes the Editor to relabel all the local labels in the above program.

(After executing each RElabel command, you'll have to first DElete all the contents of the work file, then re-CONCATenate the program to see the effect of each RELABLE.

RE 2 <ENTER>

The Editor does not relabel any lines. It returns **ITEM NOT FOUND** followed by the line that you specified.

RE 1,2 <ENTER>

No relabeling occurs. The Editor returns **ITEM NOT FOUND** followed by the last line you specified.

RE 1-3 <ENTER>

causes the Editor to relabel GLOBAL ONE's range only.

RE 2,5 <ENTER>

causes the Editor to relabel GLOBAL ONE's range only.

RE 2-6 <ENTER>

causes the Editor to relabel GLOBAL ONE's and GLOBAL TWO's ranges.

RE 3.7 <ENTER>

causes the Editor to relabel GLOBAL TWO's range only.

RE ALL <ENTER>

causes the Editor to relabel each GLOBAL RANGE in the program.

When you RELABEL all of the program, it should be:

```
1 $1      DATAL   1Ø
2 GLOBAL ONE
```

```
3 $1       BE      $2
4 $2       BLT     $1
5 GLOBAL TWO
6 $1       NOP
7          LDA     .A0,$1
8 GLOBAL THREE
```

SAVE

SA (lines) filespec

saves filespec to disk.

lines is optional. If you omit it, the Editor saves all of the current program.

If you do not specify filespec, the Editor saves the program under the most recently CONCATenated or SAVEd filespec. Before doing so, it displays the filename, followed by "CANCEL = 'X'". To cancel the SAve, type X <ENTER>; to continue the SAVE, press <ENTER>.

Examples

With the sample program -- SAMPLE2/PRO -- CONCATenated in the work file, type:

```
SAVE SAMPLE2/PRO <ENTER>
```

The Editor SAVEs the program to disk, writing over any existing file with the same name.

To SAVE only the first two lines of the program, type:

```
SA (1,2) SAMPLE2/PRO <ENTER>
```

If you type:

```
SA <ENTER>
```

the Editor prompts you with:

```
SAMPLE2/PRO

CANCEL = 'X'
```

If you want to discontinue the SAVE, type X <ENTER>; if you want to continue the SAVE, press <ENTER>.

SEARCH

SE (lines)delimiter string delimiterG

causes the Editor to search for string within the range of lines.

lines is optional. If omitted, the Editor SEarches the current line only. You must enclose lines in parentheses ( ).

G is optional. When used, it tells the Editor to list all occurences of string within the range of lines. If not included, the Editor lists only the first occurrence of string.

Examples

CONCATenate SAMPLE2/PRO into the empty Editor (every line has an absolute line number), and type:

```
SE (1-2)*LOOK* <ENTER>
```

"*" is the delimiter. The Editor searches lines 1 and 2 for the string LOOK. Since you didn't specify G, the Editor displays only the first line containing LOOK.

```
SE (ALL)*BNE*G <ENTER>
```

The Editor searches all lines of the program for the string BNE and displays every line containing it followed by the current line.

STRING

ST (lines)delimiter string1 delimiter string2 delimiter G

searches lines of the program for string1 and replaces it with string2.

lines is optional; if omitted, the current line is used. You must enclose lines with parentheses ( ).

G is optional. When used, it tells the Editor to substitute string1 with string2 at every occurence of string1 within the range of lines. If you don't specify G, the Editor substitutes string2 for string1 at the first occurrence within the range of lines.

string1 is optional. If you omit it, the Editor inserts string2 at the beginning of every line within the range of lines.

string2 is optional. If you omit it, the Editor deletes string1.

Examples

With SAMPLE2/PRO still CONCATenated in the Editor's work file, type:

```
ST(1-4)/THIS/THAT/G <ENTER>
```

"/" is the delimiter. The Editor finds all occurrences (G is specified) of the string THIS and substitutes each with THAT.

By typing:

```
ST (ALL)/THAT/THIS/ <ENTER>
```

the Editor finds all occurrences of the string THAT and replaces it with THIS, and stops at the first occurrence (no G specified).

```
ST/E// <ENTER>
```

causes the Editor to search the current line for the string E. Since the second string is empty, it deletes E.

TAB

TA c,tabstop1,tabstop2,....

allows you to set tabs for use in the Insert mode (I? prompt).

c can be any character that you choose to represent the tab. (except blank or $). If you don't specify c, the Editor clears all tab stops.

tabstop1, tabstop2, ... are the positions where you set the tabs stops. You can specify up to eight tab stops, separating each with a comma. The first tab in a line is tabstop1, the second tabstop2, etc.

If you specify only the tab character, the Editor keeps all previous tab stops. In this way, you can change the tab character without altering the actual tab stops. This is especially useful when c is a character used in the text of an insert.

To set a tab (at the Editor's commmand mode -- C?), type TA followed by the tab character, a comma, then a string of tab stops (column number where you want to place tabs). You can enter up to eight tabs.

You can also use the <TAB> key instead of the TABS command. Its tabstops are preset to multiples of eight (i.e., 8, 16, 24, 32, etc.).

When you enter <TAB> in the Insert mode, it displays a ± character. In the Command mode, the Editor displays the actual tabbed spacing.

Use the <TAB> key when you want the multiple of eight tabstops; otherwise set your own tabstops with the TAB command.

When setting tab stops:

1. If you don't specify any variables, tab stops are cleared. For example, TA <ENTER> clears all tab stops.

2. If you specify only the tab character, the actual tab stops remain the same.

3. The tab character can be any character except blank, $, or one included in the text.

4. Remember where you set the tab stops because, when you use them in the Insert mode, the Editor does not display the actual tab spacing.

5. Always set the maximum number of tabs that you'll want to use. For example, if you only set three tabs with the TA command, you can only reference three in the Insert mode. If you reference more than you set, the Editor deletes any information following the extra tab(s).

You can use a tab character at any time in the Insert mode. For example, set tabs at 1Ø, 2Ø, and 35, with a slash (/) as the tab character, by typing:

```
TA /,1Ø,2Ø,35 <ENTER>
```

Now you can use tabs in any lines you type in the Insert mode. To do so, simply type the slash (/) before the word or phrase you want tabbed. You can use as many tabs as you set with the TAb command. Type:

```
BEGIN/LD/.AØ,#TABLE/*load start of table <ENTER>
```

Now the program line is set with tabs. Whenever the Editor displays this line, it does so with the contents of the line tabbed like this:

```
BEGIN    LD        .AØ,#TABLE    *load start of table
```

Examples:

```
TA/,5,12,3Ø <ENTER>
```

Sets tabs at columns 5, 12, and 3Ø. The tab character is the / (slash).

```
TA ? <ENTER>
```

Changes the tab character to ? (question mark) and keeps all previous tab stops (5,12, and 3Ø) the same.

TA <ENTER>

Clears all tabs.

CHAPTER 3

THE ASSEMBLER-16

CHAPTER 3/ THE ASSEMBLER-16

The Assembler-16 assembles a source file into an intermediate, relocatable object code file.

This Chapter explains how to operate the assembler. For information on the source format required by the Assembler, see Section II.

THE ASSEMBLER COMMAND

This command, typed at TRSDOS-16 Ready:

ASM16 source filespec {options} comment

loads the Assembler-16 and assembles the source filespec using the default options or the options you specify. (The options and default options are described below).

If you omit the source filespec's extension and **include the disk identifier** (drive or disk name), the Assembler-16 uses the disk identifier instead of the extension.

If you omit the source filespec's extension and also **omit the disk identifier**, the Assembler-16 uses /SRC as the extension.

You can use a source file produced by other editors, as well as the Assembler-16's Editor, provided it is in one of these formats:

(1) variable length record (VLR) with one statement per record preceded by a byte count (this is what the Editor in this package produces)
(2) ASCII stream with one byte per record (an LRL of one) and each line terminated by a carriage return byte.
(3) fixed length record (FLR) with one statement per record.

The optional comment allows you to document the assembly. If you use the comment, you must enclose the options in braces {}. Otherwise, the braces are not required.

ASSEMBLER OPTIONS

You can specify the one letter options in one of four ways:

option	switches the option ON
option=Y	switches the option ON
option=N	switches the option OFF
option=drive	switches the option ON using the specified drive

You can separate each option with a comma, a space, or nothing (not separate them at all).

The options and their defaults (what the Assembler-16 uses if you omit the option) are:

C (Current Record Count)

Displays the number of the record currently being assembled. The default is C=N.

E (Errors Only)

Produces a listing of only the records which generate an error, along with any any object produced, and the error message. The default is E=N.

K (Keep Work Files)

Keeps the Assembler-16 from deleting the work files. (See the W switch for an explanation of the work files.) This speeds up multiple assemblies, since the Assembler-16 will not have to create work files over and over again. The default is K=N.

L (Listing File)

Creates a listing disk file using the same filename as the source, with the extension /LST. The default is L=N.

O (Object File)

Generates an object file using the same filename as the source, with the extension /OBJ. The default is O=Y.

P (Print Listing)

Prints the listing on the printer. The default is P=N.

S (Short Listing)

Truncates listing if the source is too long for a line (as opposed to wrapping-around). The default is S=N.

T (Terminal Listing)

Prints the listing on the video display terminal. The default is T=Y.

U (Uppercase Conversion)

Converts lowercase letters to uppercase letters. This is useful for printers without lowercase. The source remains unchanged. The default is U=N.

W (Work File Specification)

Allows you to specify which drive the Assembler-16 should use for its work files. The default is W=Ø. W=N is not allowed.

To save memory, the Assembler-16 creates two work files during the assembly for temporary storage. It names the files:

```
ASMSØØØp/WRK
ASMPØØØp/WRK
```

(p is the partition number for multi-tasking environments. Multi-tasking will be available in a future release of TRSDOS-16. In a single-tasking environment, p=1.)

The Assembler-16 stores this file in your primary drive (unless you use W=drive.) When it finishes the assembly, it deletes the work files (unless you use K=N).

EXAMPLE ASSEMBLER COMMAND

This command, typed at TRSDOS-16 Ready:

ASM16 FRED {C=Y,L,P,U,W=4,K} <ENTER>

causes the Assembler-16 to assemble the source file named FRED/SRC and:

- display a current record count on the screen (C=Y)
- generate a listing file named FRED/LST (L)
- generate an an object file named FRED/OBJ (the default of O is O=Y)
- print the listing on the printer in all uppercase (P,U)
- store the work files on drive 4 (W=4)
- retain (not delete) the work files (K)

THE ASSEMBLER LISTING

1. Side-by-Side Listing Format

The side-by-side listing starts on a new page. These are two examples of it:

Example 1:

```
8234C ØØE12F      43F8 743 *       LDA     .A1, /MIT25 LABEL4
                  Ø6E4+
```

Example 2:

```
6ØØ   ØØØ586      227C             LDL     .A1,#3
               ØØØØØØØ3
```

The meaning of the examples is as follows:

Line 1

columns 1-4	contains the source file line number in decimal notation (Example 1 is line 8234; Example 2 is line 6ØØ)
column 5	tells whether the line is from the primary source filespec or a file which was

	copied into it. (See the COPY directive in Chapter 9): . a blank space indicates the line is from the primary file (Example 2) . the characters A-I indicate the line was copied. (Example 1 indicates the line was part of "C", the third file copied into the program.)
column 7-12	contains the hexadecimal memory address where the operation word is stored. (Example 1 indicates the LDA instruction is at H'E12F; Example 2 indicates LDL is at H'Ø586.)
column 14-17	contains the hexadecimal operation word. (The lines immediately below it reflect the extensions.) (Example 1 indicates the operation word for LDA is 43F8; Example 2 indicates LDL is 227C)
columns 19-22	tells where the symbol used in the line (if any) was defined. (Example 1 indicates MIT25 LABEL4 was defined in line 743.)
column 23	tells whether the symbol used in this line is from the primary file or copied. . a blank space indicates it was from the primary file (Examples 1 and 2 both contain blank spaces in this column) . the letters A-I indicate the symbol was from a copied file.
column 24	tells if the symbol used in the line is a backwards reference (defined in a previous line): . an asterisk (*) indicates it is a backwards reference . a blank space indicates it is a forwards reference (Example 1 indicates MIT25 LABEL4 was defined in a previous line.)
column 26-end	is the source line, printed with no reformatting. If the source line is long, the excess is printed right justified on the following line.

Line 2

columns 1Ø-17 is the extension of the operation word, right justified (Example 1 indicates the extension for the instruction is Ø6E4; Example 2 indicates a long extension of ØØØØØØØ3.)

column 18 is the relocation type of the symbol used in the line (if any):

- a period (.) indicates it is external (defined by DEF)
- a plus sign (+) indicates it is relocatable (defined in an RSECT)
- a blank space indicates it is absolute (defined in an ASECT)

(Example 1 indicates the symbol MIT25 LABEL4 is relocatable; Example 2 does not contain a symbol) See Chapter 9 for an explanation of the DEF, RSECT, and ASECT directives.)

The Assembler-16 may print line 2 one to four times, depending on how many extensions the operation word uses.

2. Error/Warning Messages

The error and warning listing begins immediately after the side-by-side listing. These are examples of it:

Example 1:

```
        2  ØØØØØØ ØØØØØØØØ    1          LOAD      .Dl, #32
                                                $
*** ERROR *** 1 UNKNOWN OPCODE
*** ERROR *** 1 ILLEGAL STATEMENT
```

Example 2:

```
        3  ØØØØØ4     3Ø3C               LD        .BlXYZ, #32
                         ØØ2Ø

*** ERROR ***   REGISTER SYMBOL REQUIRED BlXYZ
** WARNING **   WORD LENGTH ASSUMED
```

Example 3:

```
        4  ØØØØØ8      323C                  LD        .D1, #32
                            ØØ2Ø

** WARNING **   WORD LENGTH ASSUMED
```

The meaning of each line is as follows:

Line 1 is the line containing the error

Line 2 indicates with a dollar sign ($) the position of the errors or warnings. The Assembler-16 uses this notation only where it is helpful.

Line 3 is the error message. Line 3 is repeated for each error in the source line. This is the meaning of each column in line 3:

column 1-13 tells the type of message:

- *** ERROR *** (this causes the program not to assemble properly)
- ** WARNING ** (this warns you that the Assembler-16 might not be interpreting your instruction as you want it to)

column 15-16 identifies which $ character the error message is referencing (Both messages in Example 1 indicate that they are referencing the first -- actually the only -- $ character)

columns 18-end contains the error/warning message, followed by a blank space, followed by the associated symbol (if any). If the symbol is longer than space permits, it is truncated.

List of Errors and Warnings

CONSTANT OUT OF RANGE
ILLEGAL BINARY CONSTANT
ILLEGAL COPY STATEMENT
ILLEGAL EXPRESSION
ILLEGAL FORMAL
ILLEGAL HEX CONSTANT
ILLEGAL LOCAL
ILLEGAL OCTAL CONSTANT
ILLEGAL STRING DELIMITER
ILLEGAL SYMBOL
MISSING COMMA
COPY FILE NOT FOUND
FORM FIELDS SIZE ERROR
ILLEGAL ORG EXPRESSION
INCORRECT NUMBER OF EXPRESSIONS
INDEXING NOT ALLOWED
INVALID LOCAL
LOCAL LABEL NOT ALLOWED
NESTED COPY NOT ALLOWED
OPERAND INCOMPATIBLE WITH INSTRUCTION
REGISTER SYMBOL REQUIRED
STATEMENT IGNORED

MISSING COMMENT SEPARATOR
MISSING SEPARATOR
SYMBOL REQUIRED
UNKNOWN OP CODE
MISSING RIGHT PAREN
MISSING STRING TERMINATOR
PACKED STRING NOT ALLOWED
GLOBAL SYMBOL REQUIRED
INVALID FORMAL REFERENCE
INVALID GLOBAL SYMBOL
LOCAL MULTIPLY DEFINED
LOCAL UNDEFINED
SYMBOL MULTIPLY DEFINED
SYMBOL UNDEFINED:
ABSOLUTE EXPRESSION REQUIRED
ADDRESS REGISTER REQUIRED
SYMBOL NOT POP
TOO MANY COPY STATEMENTS
USE OF ILLEGAL FORM SYMBOL
VALUE NOT RELATIVE TO CURRENT PSECT
VALUE OUT OF RANGE
VALUE TRUNCATED
BIT NUMBER OUT OF RANGE
SHIFT VALUE OUT OF RANGE
SMALLER LENGTH ATTRIBUTE ASSUMED
WORD LENGTH ASSUMED
DIVISION BY ZERO
ILLEGAL STATEMENT
REPEATED RLIST ELEMENT - IGNORED
INCORRECT NUMBER OF OPERANDS
DSECT SYMBOL REQUIRED
RSECT SYMBOL REQUIRED

3. Cross Reference Listing

The cross reference listing starts on a new page. This is an example of it:

```
LABELSCANHAVESINGLESPACESBUTDONTSHOWTHEM      ØØØØ3Ø2 L+  39ØC
                          331 /MOV   336 /LD    337 /STP    338 /XCH
                          34Ø /LDL   343 /ADD   344 /LD     356 /MOV
                          376 /STB
```

The meaning of it is as follows:

Line 1

columns 1-45 is the first 45 nonblank characters of the symbol, with trailing blanks.

column 56 is the length attribute of the symbol:
- L = Long
- W = Word
- B = Byte
- U = Undefined

column 57 indicates where the symbol's relocation type:
- blank space = absolute (defined in a program section initialized by the ASECT directive)
- + = relocatable (defined in a program section introduced by the RSECT directive)
- . = external (declared by the DEF directive)

(If the program contains none of the above

	directives, the Assembler-16 treats all the symbols as absolute.)
columns 59-62	contains the source line number where the symbol is defined (**** indicates the symbol's undefined.)
column 63	tells whether the symbol is from the primary source filespec or a copied file: blank space = primary file A-I = copied file

Lines 2-n

columns 3Ø-33	contains the source line number where the symbol is referenced.
column 34	tells whether that line is from a primary or copied file: blank space = primary file A-I = copied file
columns 35-4Ø	contains the source line's instruction preceded by a slash (/)

The Assembler-16 lists line 2 as many times as there are references to the symbol. If there are no references, the Assembler-16 does not list line 2. Columns 3Ø-41 are repeated across the entire width of the page.

4. Statistics Listing

The statistics listing is the final page of the listing. It lists the total ERROR and WARNING messages (in decimal notation).

Example:

```
STATISTICS OF THIS ASSEMBLY

TOTAL NUMBER OF ERRORS            6
TOTAL NUMBER OF WARNINGS          8
```

CHAPTER 4

THE LINKER

CHAPTER 4/ THE LINKER

The Linker (LINK16) links one or more "intermediate" files into an absolute program that the 68000 can execute. It does this in one pass.

In addition, the Linker can load multiple object files, resolve undefined external references between the modules, and produce a single program file.

PREPARING A LINKER CONTROL FILE

Before using the Linker, you must prepare a Linker control file by:

1. Creating an object file, and then
2. Creating the control file

1. Creating an Object File

The object file is a file of intermediate, relocatable object code. You will normally use the Assembler-16 to create it.

You can also create it with another assembler or compiler provided it produces the format required by the Linker. Appendix A describes the format.

2. Creating the Control File

The control file is a file of linker directives. You will normally use the Editor, as demonstrated in Chapter 1 to create it.

You can also use other editors to create the file, provided it is in the format required by the Linker:

(1) variable length record (VLR) with each record preceded by a byte count (this is what the Editor in this package produces.)

(2) ASCII stream with one byte per record (an LRL of one) and each line terminated by a carriage return byte.

(3) fixed length record (FLR) with one record per statement.

The directives you can use in creating this file are:

END

END

ends the control file. If the Linker reaches the end of the file and does not encounter an END directive, it automatically supplies one.

Example:

```
END
```

tells the Linker to stop reading the control file, finish producing the absolute program file, and return to TRSDOS-16 Ready.

INCLUDE

INCLUDE <u>object filespec</u>

inputs the <u>object filespec</u> and links it to the existing program.

If you omit the <u>object filespec</u>'s extension, the Linker appends the extension /OBJ. However, if you omit the extension and include the drive, the Linker treats the drive as the extension.

Example:

```
INCLUDE FILE1
```

causes the Linker to load FILE1/OBJ and link it to the current program.

```
INCLUDE FILE1:2
```

causes the Linker to load FILE1/:2 and link it to the current program.

ORIGIN

ORIGIN address

forces the next INCLUDEd filespec to be loaded beginning at the specified address. The Linker assumes this address is decimal unless it has either a leading '0' or '>', both of which imply hexadecimal.

The Linker will not test for conflicts with previously loaded code. For example, if you specify the same ORIGIN address for two filespecs, the Linker will load one on top of the other without giving you a warning.

If you omit ORIGIN, the Linker uses an originating address of ØØØØ.

Examples:

```
ORIGIN 5ØØØ
```

tells the Linker to use decimal 5ØØØ as the originating address for the next INCLUDEd file.

```
ORIGIN >1ØØØ
```

tells the Linker to originate the next file at hexadecimal 1ØØØ.

*****

```
*
```

Begins a comment line. The Linker will print it on the map, but otherwise ignore it.

Example:

```
*This is a comment
```

is ignored by the Linker.

THE LINKER COMMAND

This command, typed at TRSDOS-16 Ready:

LINK16 control filespec {options} comment

loads the Linker, executes the directives in the control filespec, and produces an absolute program file. The absolute program file will have the same name as the control filespec, minus the extension.

If you omit the control filespec's extension, the Linker appends the extension /CTL. However, if you omit the extension and include the drive, the Linker treats the drive as the extension.

LINKER OPTIONS

As with the Assembler-16, you can specify the one letter options in one of four ways:

option	switches the option ON
option=Y	switches the option ON
option=N	switches the option OFF
option=drive	switches the option ON using the specified drive number (for the M and O options only)

The options and their defaults are:

L (Create Map File)

Creates a file containing a Linker map on the drive specified. The Linker assigns this file the same filename as the source, with the extension /MAP. The default is M=N.

O (Output Program File)

Creates a final, executable program file on the drive specified. The default is O=Y.

P (Print Linker Map on Printer)

Prints the Linker map. The default is P=N.

T (Print Linker Map on Terminal)

Prints the Linker on the video display terminal. The default is T=Y.

EXAMPLE LINK

This is an example of linking a control file named TEMP/CTL. The next section, "The Linker Map" will use it to demonstrate the the various maps the Linker outputs.

TEMP/CTL (created with the Editor) contains these directives:

```
*LINK OF TEMP/CTL
INCLUDE TEMP/OBJ
ORIGIN Ø2ØØ
INCLUDE TEMPB/OBJ
ORIGIN Ø5ØØ
INCLUDE TEMPC/OBJ
INCLUBE THREE/OBJ
END
```

Note: The word INCLUBE (in the next to the last line) is intentionally misspelled. The section on "The Linker Map" uses this misspelling to demonstrate how the Linker outputs errors.

This command, typed at TRSDOS-16 Ready:

```
LINK16 TEMP {L=2,P,T=N} <ENTER>
```

causes the Linker to link an absolute program, following the directives in TEMP/CTL, and:

- create a map file on drive 2 named TEMP/MAP (M=2)
- print the map on the printer (P)
- not print the map on the video terminal (T=N)
- create a program file named TEMP (O=Y is the default)

THE LINKER MAP

The Linker map consists of the following:

1. linker control listing

2. allocation map
3. definitions map
4. undefined references map
5. summary

1. Linker Control Listing

The Linker outputs the linker control listing first. It contains each directive from the control filespec interspersed with error messages.

The above example produces the following listing:

```
ØØØØ1      * LINK OF TEMP/CTL
ØØØØ2      INCLUDE TEMP/OBJ
ØØØØ3      ORIGIN Ø2ØØ
ØØØØ4      INCLUDE TEMPB/OBJ
ØØØØ5      ORIGIN Ø5ØØ
ØØØØ6      INCLUDE TEMPC/OBJ
ØØØØ7      INCLUBE THREE/OBJ
*****      $ - ILLEGAL COMMAND
ØØØØ8      END
*****      WARNING - UNRESOLVED EXTERNAL REFERENCE AT ØØØ5Ø2
```

The Linker prints error messages as it encounters them. There are three classes:

ERRORS -- these are caused either by syntax errors in the control file or object code errors. (The above example has an ILLEGAL COMMAND error due to the misspelling of INCLUBE.)

WARNINGS -- these are to advise you that the Linker might be interpreting your file in a different way than you intended. (The warning in the above example is due to an undefined symbol, MYSTERY SYMBOLIC LABEL, which appears in the undefined references map.)

FATAL -- these are caused by errors in the object code construction. You should never get one of these errors when using the Assembler-16. These errors will always cause the Linker to abort the linkage. Note that an error line may be printed immediately before the fatal diagnostic.

Note that the above example lists the comment, *LINK OF TEMP/CTL. However, this comment has no effect on the linkage.

2. Allocation Map

The allocation map describes where in memory the Linker has located the INCLUDEd object files (modules).

The above example produces this map:

ALLOCATION MAP

MODULE	NO	ORIGIN	LENGTH	DATE	CREATOR	VER
TEMP	1A	ØØØØØØ	------	11/24/81	ASM-16	1.Ø
TEMPB	2A	ØØØ2ØØ	------	11/24/81	ASM-16	1.Ø
TEMPC	3	ØØØ5ØØ	ØØØØØC	11/24/81	ASM-16	1.Ø

The meaning of each column is as follows:

MODULE -- the linked object filename. Only the filename (not the extension) is printed.

NO -- the sequential order in which the modules are INCLUDEd. If the module is part of an ASECT (an absolute program section), this number is followed by the letter A.

ORIGIN -- the memory address where the module is loaded.

LENGTH -- the length of the module. The characters '------' in this column indicate the module is part of an ASECT and therefore the length is undefined.

DATE -- the date the module was created.

CREATOR -- the name of the assembler or compiler used to create the module.

VER -- the version number of creator (1.Ø for ASM-16).

The above example indicates that the Linker included TEMP/OBJ, an absolute section, at address ØØØØ.

TEMPB/OBJ and TEMPC/OBJ, both relocatable, are included at hexadecimal 2ØØ and 5ØØ. THREE/OBJ is not included, since the word INCLUBE was misspelled.

3. Definition Map

The definition map lists all the symbol defined as external by the DEF directive.

The above example produces this map:

```
                    DEFINITIONS

     SYMBOL                                   VALUE      PROG
SYMBOLICLABEL                                 ØØØØ52        1A
```

The meaning of each column is as follows:

SYMBOL -- the symbol itself. A maximum of 45 characters are printed. (This example contains only one.)

VALUE -- the symbol's value

PROG -- the module which defined the symbol. If the number is part of an ASECT, the letter A follows it.

The Linker does not sort this map. It prints the symbols in the same order it encounters them.

The Linker will not print this listing if there are no externally DEFined symbols.

4. Undefined Reference Map

This is a listing of all symbols referenced, but not defined.

The above example produces:

```
                    UNDEFINED REFERENCES

     SYMBOL
MYSTERYSYMBOLICLABEL
```

indicating the file references only one undefined symbol -- MYSTERYSYMBOLICLABEL -- which was what caused the warning in the directive listing.

The Linker prints the symbols in the same order it encounters them. It does not print the map if there are no undefined symbols.

5. Summary

This is the completion message, with a count of errors and warnings.

The above example produces:

```
RSECT LENGTH    = ØØØØØC
PROGRAM ENTRY   = ØØØØØØ
PROGRAM LENGTH  = ØØØ5ØC
POS INDEPENDENT = NO

LINK COMPLETE: ØØØØ1 ERRORS, ØØØØ1 WARNINGS.
```

which shows that ØØØC is the length of TEMPC/OBJ, the one relocatable program section; ØØØØ is the originating address of the entire program; Ø5ØC is the length of the entire program; and the program is not position independent..

The Linker generated one error and one warning.

ERROR MESSAGES

This is a listing of the Linker error messages, divided into three groups:

1. ERRORS
2. WARNINGS
3. FATAL

ERRORS

These messages alert you that the program will not link properly.

FILE UNAVAILABLE, CODE= <u>error code</u>

The file included in the control file cannot be accessed for the reason cited by the TRSDOS error code.

For example:
     FILE UNAVAILABLE: 'filename', CODE = 24
indicates TRSDOS error 24 (file not found).

ILLEGAL COMMAND
The directive is not one of those allowed by the Linker.

SYNTAX ERROR
The directive has not been typed correctively.

WARNINGS

These messages warn you that the Linker might be taking a different action than you intended.

DOUBLE DEFINED RSECT
The module has more than one RSECT (for example, a blank RSECT and a named RSECT). The Linker allows only one RSECT per module.

DOUBLE DEFINED SYMBOL: 'symbol'
The symbol enclosed in quotes is defined in two separate modules. The Linker uses the first definition it encounters.

ERRORS GENERATED DURING ASSEMBLY
There were errors generated during the assembly of the module.

SIZE ERROR AT: absolute address
An external will not fit into the field reserved. The Linker lists the absolute address of the modifiable part of the instruction generating the error.

For example:
     SIZE ERROR AT: ØØØ2Ø2
might occur when address 2ØØ of the module contains an instruction such as:

```
     LDW                 .DO,/external
```

with the Linker evaluating 'external' as greater than H'FFFF (the maximum value that will fit into a word).

To solve the problem, put a '.L' after the name of the external wherever it is referenced. For example:

```
                LDW       .DO,/external.L
```
will never generate a SIZE ERROR. (However, it does cost an extra word of program storage over the previous example.)

Note that the address associated with the size error does not indicate the address of the instruction. Instead, it indicates the address of the modifiable field (the address where the external will be loaded.)

UNRESOLVED EXTERNAL REFERENCE AT: absolute address
The Linker did not find a definition (DEF) for an external reference (REF). The Linker lists the absolute address of the instruction generating the error.

WARNINGS GENERATED DURING ASSEMBLY
There were warnings generated during the assembly of the current module.

FATAL

These errors should NEVER occur in normal use of the Linker. The Linker detects these errors primarily to support development of compilers and assemblers.

If one of these errors occurs while using the Assembler-16, it should be considered a BUG and be reported. See also the OBJECT CODE DESCRIPTION section in this manual.

Note that unless otherwise stated, these errors are FATAL. The Linker will abort with an error message.

DOUBLY DEFINED SECTION LENGTH
The Linker encountered two DEFINE SECTION LENGTH plexes in the object stream.

END OF OBJECT ENCOUNTERED
The object code stream was improperly terminated (no END OF OBJECT plex). This may be caused by terminating the assembler (with BREAK key) before normal completion.

ILLEGAL POLISH EXPRESSION
An object code expression was encountered that does not conform to the standards set forth in the object code description.

MODULE HAS MORE THAN ONE PROCESSOR DEFINED
The DEFINE PROCESSOR plex was encountered more than once.

UNDEFINED EXTERNAL SYMBOL
A SYMBOL REFERENCE was made without that symbol previously being defined.

UNDEFINED OPERATOR
An invalid operator was encountered during expression evaluation.

UNDEFINED POLISH COMMAND
An invalid operand was encountered in expression evaluation.

UNDEFINED RSECT SELECTED
An attempt was made to select a section (RSECT) that was not previusly defined (opened).

RSECT LENGTH UNDEFINED
A define section length plex was not encountered in the object stream. This is only a warning (the linker does not abort).

CHAPTER 5

THE DEBUGGER

CHAPTER 5/ THE DEBUGGER

The Model 16 Debugger (DEBUG) allows you to:

. debug an existing 68ØØØ machine code program
. insert a 68ØØØ machine code program into memory

The way to start the Debugger depends on which you want to do.

(DEBUG will not load if it is not included in a file named CONFIG16/SYS. See Appendix D if it does not load properly.)

STARTING THE DEBUGGER

To Debug an Existing Program...

To debug an existing program, type (at TRSDOS-16 Ready):

DEBUG ON <ENTER>

which turns ON a switch causing the Debugger to activate. While this switch is ON, any program you load is loaded into the Debugger. (The Debugger remains dormant until you load a program.)

For example, with the DEBUG ON, type:

SAMPLE <ENTER>

which loads SAMPLE into the Debugger. The Debugger displays the Register Display and a # prompt.

The # prompt indicates you are in the Debugger command mode and can enter any of the Debugger commands. Type:

H <ENTER>

for a menu of all the commands.

To turn OFF the Debugger switch, type (at the Debugger # prompt):

O <ENTER>

or (at the TRSDOS-16 Ready prompt):

DEBUG OFF <ENTER>

To Insert a New Program...

To insert a new machine-code program with the Debugger, type (at TRSDOS-16 Ready):

DEBUG <ENTER>

which causes the Debugger to activate, displaying the Register Display and the # command prompt.

To insert a program beginning at address 5ØØØ, type:

C 5ØØØ <ENTER>

the Debugger displays address 5ØØØ (in parentheses and its contents. To enter the MOVW instruction, type:

3ØBC <ENTER>

and the Debugger inserts 3ØBC, the operation word for this instruction, and waits for you to insert the next instruction. (See Chapter 8, "Instructions", for information on machine code operation words.)

Type Q <ENTER> to return to the # prompt. Type Q <ENTER> again to exit the Debugger.

REGISTER DISPLAY

When you first start-up the Debugger, the Register Display appears on your screen. Certain Debugger commands will "call" it (cause it to appear again) or update it.

This is an example Register Display:

```
TRS-8Ø Model 16 DEBUG Version 3.Ø
PC=ØØØØØØR  X=Ø   N=Ø   Z=Ø   V=Ø   C=Ø   IM=Ø  S=U
A=ØØ8Ø18FE  ØØ8ØØA4Ø  ØØ8Ø1B4E  ØØ8Ø1B6A  ØØØØØØØØ  ØØ8Ø14EE  ØØ8Ø1886  ØØ8Ø2ØØØ
D ØØØØØØE8  ØØØØØØØØ  ØØØØFFFF  ØØØØØØØØ  ØØØØØØØØ  ØØØØØØØØ  ØØØØØØØØ  ØØØØØØØØ
```

Line 1 contains the contents of the PC (program counter register and and the value of the condition codes. Notice that the PC register is set to a "relative" address. (The character R indicates the address is relative.)

TRSDOS-16 loads DEBUG in memory that is "invisible" to the user. Therefore, a program origin address of zero is actually a "relative zero". All addresses are relative to the end of TRSDOS-16 and DEBUG. (See Appendix B, "Memory Map.)

Line 2 contains the contents of each of the eight address registers. (column 1 is register AØ, column 2 is A1, etc.)

Line 3 contains the contents of each of the eight data registers.

(The Debugger uses the same register notations as the Assembler-16. See Section II for a listing of these notations.)

You can display or change the contents of any of the registers. To do this, type the name of the register preceded by a period (.).

For example, at the # prompt, type:

.AØ <ENTER>

and the Debugger displays the contents of register AØ. If you do not wish to change this, simply press <ENTER>. If you do wish to change it, type the new value and then press <ENTER>.

DEBUGGER COMMANDS

To execute a Debugger command, type the one-letter command followed by <ENTER>.

Some of the commands allow you to specify parameters. You must type a blank space between the command and the parameter.

The parameters you can specify are:

. <u>value</u>

- register (register direct)
- address

Specifying a value

To specify a value, you can use:

- a number
  (must be hexadecimal)
- an ASCII character
  (must be enclosed in quotes)

or an expression of values separated by these operators:

- addition (+)
- subtraction (-)

For example, when using the Change command, it prompts you for a value. You could enter:

2121 <ENTER>

to insert the hexadecimal value of 2121,

'AB' <ENTER>

to insert the ASCII codes for 'A' and 'B', or

32+'A' <ENTER>

to insert 32 plus the ASCII code for 'A'.

Specifying a register directly

To specify a register directly, precede the register name with the character @. This indicates direct register addressing. (This is the opposite of the Assembler-16's use of the @ notation.)

For example:

D @A1 <ENTER>

displays the contents of register A1. (If A1 contains 1111, the Debugger displays 1111.)

Specifying an address

To specify an address, you can use:

. a value
(as defined above)
. a indirect register, preceded by a period (.)
(the Debugger interprets the period as **indirect** addressing.)

For example:

D 1ØØØ <ENTER>

displays the contents of address 1ØØØ.

D .A1 <ENTER>

displays the contents of the address contained in register A1. (If A1 contains 1111, the Debugger displays the contents of address 1111.)

D 1ØØØ+5Ø

displays the contents of address 1Ø5Ø.

When you specify an address, the Debugger assumes you mean an address relative to base zero. You can change this assumption by changing the value of the displacement register (with the R command), and then specifying the address with the letter 'R'.

For example, if you set the displacement register to a relative 2ØØØ:

D 1ØØØR

displays the contents of address 3ØØØ. The Debugger computes this address as the sum of 1ØØØ and 2ØØØ, the contents of the displacement register.

D 1ØØØ+5ØR

displays the contents of address 3Ø5Ø.

The Debugger commands are:

A (Address Stop Command)

A address1,address2,mask
A address1,register,mask

Executes the program being debugged until the contents of address2 or register is changed.

You can specify the size of address2 or register with:
/B (byte)
/W (word)
/L (long word)

address1 is optional. If specified, execution begins at address1. Otherwise, execution begins at the current address.

mask is also optional. It allows you to mask address2 or register.

Examples:

```
A ,@A5
```

executes the program from the current position until register A5 is changed.

```
A ,@A5,FFØØ
```

stops execution when the third and fourth bytes (specified by the mask) of register A5 is changed.

```
A 5ØØ,1Ø44/L
```

executes the programming beginning at address Ø5ØØ until the long word at 1Ø44 is altered.

B (Breakpoint Command)

B breakpoint, address
B breakpoint
B breakpoint/

Allows you to:

- set up to eight breakpoint addresses
- display the contents of a breakpoint
- reset a breakpoint

Be sure to set the breakpoint address at the beginning of an instruction -- never in the middle of an instruction.

For example:

B 1,1ØØØ <ENTER>

sets 1ØØØ as the first breakpoint address. When you execute your program (with the G command), it will stop executing when it reaches address 1ØØØ.

B 1 <ENTER>

displays breakpoint 1. You can enter a new value or simply press <ENTER>. (******** means the breakpoint is not set.) Both the relative and non-relative values are displayed.

B 1/ <ENTER>

resets breakpoint 1 (to 32 bit-1).

B <ENTER>

displays all the breakpoints currently set.

C (Change Command)

C address

Enters the "change mode" displaying the contents of the specified address (in parenthesis), followed by its contents.

Type value <ENTER> to insert a new value for that address. (value can be one to four bytes.)

Type / <ENTER> to see the contents of that address again.

Press <ENTER> to see the contents of the next address (the next memory word).

Press <CTRL> <9> <ENTER> to see the contents of the previous word (the previous memory word).

Type Q <ENTER> to exit the "change mode".

Example:

    C FØ6A

displays the contents of address FØ6A.

    3Ø7A <ENTER>

changes the contents of that address to 3Ø7A.

    / <ENTER>

displays the address with its new contents.

    <ENTER>

displays the next address.

    <CTRL> <9> <ENTER>

displays the previous address.

    Q <ENTER>

exits the change mode.

D (Display Command)

D <u>address</u> or <u>register</u>,<u>address</u> or <u>register</u>,<u>address</u> or <u>register</u>

calls the Register Display and displays the contents of up to three address or registers in the top right-hand corner.

The Debugger will continually update this display as you debug the program.

Examples:

D A4ØB, .A5, @A4

displays the Register Display with the contents of address A4ØB, the address specified by register A5, and the contents of register A4.

D @A2 <ENTER>

displays the contents of register A2.

E (Erase Breakpoints Command)

E

Erases the breakpoints.

G (Go Command)

G address

executes the program beginning at the specified address. Execution continues until the Debugger reaches a breakpoint (set with the B command) or the end of the program.

address is optional. If omitted, execution begins at the current address.

Example:

G 4Ø2B <ENTER>

executes the program being debugged at address Ø4Ø2B and continues until the Debugger encounters a breakpoint or the end of the program.

H (Help Command)

H

Displays all the Debugger commands.

N (Next Instruction Command)

N

executes the next instruction and then calls the Register Display.

After entering the N command, simply press <ENTER> to execute another "next" instruction. Executing any other command exits this "next instruction execution mode".

If the next instruction is a call to a subroutine, the Debugger executes the entire subroutine. (Use the S command to single step through the subroutine.)

Example:

N

executes the next instruction in the program currently being debugged.

O (Quit Debug with DEBUG OFF Command)

O

Turns DEBUG OFF and exits the Debugger. The next program will load into the normal TRSDOS-16 Ready.

Q (Quit Debug with DEBUG ON Command)

Q

Exits the Debugger leaving DEBUG ON. The next program will load into the Debugger.

R (Relative Addressing Command)

R value

displays and changes (if you specify value) or displays and allows you to change (if you omit value) the contents

of the displacement register. At start-up, the value of the displacement register is zero.

The R command helps in debugging relocatable program sections. By specifying an address as Relative, the Debugger will add to this address the value of the displacement register.

Examples:

R <ENTER>

displays the value in the displacement register. Press <ENTER> to leave it unchanged. Enter a new value to change it. For example:

1ØØØ <ENTER>

causes hexadecimal 1ØØØ to be the new value of the displacement register. If you specify a relative address, such as:

G 2ØØØR <ENTER>

the Debugger will interpret this as:

2ØØØ + 1ØØØ (the value in the displacement register)

causing the Debugger to begin program execution at address 3ØØØ.

R 3333 <ENTER>

causes hexadecimal 3333 to be the new value in the displacement register.

S (Step Command)

S

executes the next instruction and calls the Register Display.

The S command is the same as N, except S will single step through a subroutine.

As with the N command, press <ENTER> to execute the next instruction. To exit the "single stepping mode", enter a new command.

Example:

S <ENTER>

executes the next instruction in the program being debugged.

V (View Command)

V address1,value
V address1,address2

displays the contents of the addresses beginning with address1 (or the current address) and continuing for the number of bytes specified by value.

If the value is larger than address1, the Debugger interprets it as address2. In this case it displays the address beginning with address1 and ending with address2.

If you specify only address1, the Debugger displays 16 bytes beginning with address1.

Examples:

V 5ØØØ,1ØØ <ENTER>

displays 1ØØ bytes of memory starting at address 5ØØØ.

V 5ØØØ,55ØØ

displays memory starting at address 5ØØØ through address 55ØØ.

V 5ØØØ <ENTER>

displays 16 bytes of memory starting at address 5ØØØ.

V <ENTER>

displays the next 16 bytes of memory.

```
V .AØ
```

displays 16 bytes of memory starting with the address specified by the contents of register AØ.

The Assembler-16 contains an easy-to-use set of assembly language mnemonics for developing Motorola MC68ØØØ programs on the TRS-8Ø Model 16.

Please note that the Assembler-16 mnemonics are not the same as the Motorola mnemonics. This is due to a major effort among programming and engineering organizations to standardize mnemonics.

Since the Assembler-16 mnemonics and notations are different from Motorola's, you will need to use this Reference Guide to learn the Assembler-16 ones.

However, you will probably find it helpful to use these books to understand the logic of the 68ØØØ Microprocessor:

> **MC68ØØØ: 16-BIT MICROPROCESSOR User's Manual,** Motorola Incorporated, 198Ø
>
> **The 68ØØØ: Principles and Programming,** by Leo J. Scanlon: Howard W. Sams & Co., Inc., Indianapolis, Indiana, 1981.

KEY TO NOTATION

The following notation conventions are used in this section:

[ ]	-	the item within is optional.
B	-	a byte length string (8 bits).
W	-	a word length string (16 bits).
L	-	a long word length string (32 bits).
U	-	an undefined length string.
exp	-	an expression (described in chapter 2).
b	-	in place of a blank space.

The registers are represented as follows:

Ad or Dd	-	an address or data register used as a destination operand.
An	-	one of the eight address registers AØ-A7 (n specifies the register number).
As or Ds	-	an address or data register used as a source operand.
Au or Du	-	an address or data register used as an upper-bounds operand.
Dn	-	one of the eight data registers DØ-D7 (n specifies the register number).
Rn	-	any data or address register (n specifies the register number).
Ri	-	any register used as indexed register with optional .W (word) or .L (long) length specified.
Ri	-	can be either An.W, An.L, Dn.W, or Dn.L.
SP	-	the stack pointer.
SSP	-	the supervisor stack pointer register (SP in System Mode).
USP	-	the user stack pointer register (SP in the User Mode).

CHAPTER 6

68ØØØ ORGANIZATION

CHAPTER 6/ 68ØØØ ORGANIZATION

The 68ØØØ contains eighteen registers: eight data registers, eight address registers, a program counter, and a status register. Both address and data registers can be used for word and long word arithmetic operations as well as for indexing. In addition, address registers can be used for indirect addressing; and data registers can be used for byte arithmetic operations.

Data registers are 32 bits in size. Byte operands occupy the low order 8 bits, word operations the low order 16 bits, and long word operands the entire 32 bits. The least significant bit is always labeled as zero.

When one of the low order portions of a data register is used, only that low order portion is changed. The remaining high order position is not used or changed.

Address registers are also 32 bits in size. However, they do not support byte-sized operands. Depending on the operation, either the low order 16 bits (word) or all 32 bits (long word) are used. When the address register is a destination operand, the entire register is affected. In word size operations, the operands are sign extended to 32 bits before the operation is performed.

The **user stack pointer** (USP) is another 32 bit register. While you are in the user mode, it provides the stack address in specific stack operations such as PUSHA and LINK. While in the supervisor mode, you can use it as an operand.

The **system stack pointer** (SSP), also 32 bits, serves as the stack register while you are in the supervisor mode. While in the supervisor mode, you can use USP as an operand. You can never use SSP as an operand, nor can you use USP as an operand while in the user mode.

Note: TRSDOS-16 does not allow you to get into the supervisor mode.

Another 32 bit register is the **program counter.** It contains the memory address of the next sequential instruction to be executed.

The **status register** is a 16 bit register which contains certain system information as is illustrated below:

15	14	13	12	11	1Ø	9	8	7	6	5	4	3	2	1	Ø
T		S				I					X	N	Z	V	C

In this diagram:

T - contains the status of the trace mode (set when trace is on, cleared otherwise).

S - reflects the current status of the CPU (set in supervisor mode; cleared in user mode).

I - contains the level of interrupt recognized by the CPU.

Bits 8 through F make up the system byte of the status register. You can change these bytes only from the supervisor mode.

The least significant byte (bits zero through seven) of the status register is known as the "user byte" or the **"condition code register"** (CCR). You can address this byte from either the user or supervisor mode. The CCR contains information pertaining to the actions of the current instruction. The bits in the CCR are:

X - The **eXtend bit** is set when an instruction (like ADD) causes a carry at the most significant bit.

N - The **Negative bit** is set when an operation results in a negative number (i.e., the most significant bit of the operand is set in an arithmetic operation).

Z - The **Zero bit** is set when an operation results in a zero value.

V - The **oVerflow bit** is set when an operation causes overflow of the operand (i.e., the resulting number is too big for the size of the operand.

C - The **Carry bit** is set when an instruction (like ADD) causes a carry at the most significant bit.

Note: The carry and extend bits are set together during most operations.

MEMORY ORGANIZATION

With the 68ØØØ processor, data can be accessed by using byte (8 bits), word (16 bits), or long word (32 bits) operations. All words and long words begin on even-numbered addresses.

The most significant byte is stored first and the least significant byte is stored last:

15 14 13 12 11 1Ø 9 8	7 6 5 4 3 2 1 Ø
word ØØØØØØ	
byte ØØØØØØ	byte ØØØØØ1
word ØØØØØ2	
byte ØØØØØ2	byte ØØØØØ3
. . .	. . .
word FFFFFE	
byte FFFFFE	byte FFFFFF

Long Word/Register/Address Organization - 32 Bits (MSB and LSB signify the most and least significant bits, respectively):

15 14 13 12 11 1Ø 9 8	7 6 5 4 3 2 1 Ø
MSB	high order
long word Ø	. . . .
	low order LSB
	high order
long word 1	. . . .
	low order
long word 2	. . . .

Decimal Data - 2 binary coded decimal digits equals 1 byte (MSD and LSD signify the most and least significant bits, respectively):

15 14 13 12	11 1Ø 9 8	7 6 5 4	3 2 1 Ø
(MSD) BCD Ø	(LSD) BCD 1	BCD 2	BCD 3
BCD 4	BCD 5	BCD 6	BCD 7

INSTRUCTION FORMAT

When the Assembler-16 translates the 68ØØØ instructions into machine code, it arranges them so that the instruction word (16 bits) comes first, followed by any operands, which may include up to four more words. The result of this operation is stored in the destination. The source is the second operand. This is illustrated below:

15 14 13 12 11 1Ø 9 8 7 6 5 4 3 2 1 Ø
operation word (first word specs operation & mode)
immediate operand (if any, one or two words)
source mode register extension (if any, one or two words)
destination mode register extension (if any, one or two words)

The typical format of the operation word follows, where mode and register are three bit fields.

15	14	13	12	11	1Ø	9	8	7	6	5	4	3	2	1	Ø
X	X	X	X	X	X	X	X	X	X		mode			reg.	

ADDRESSING MODES

The 68ØØØ allows you to address data seven different ways:

1. **Implicit** - The operand is implied from the instruction.

2. **Register Direct** - The operand or operands are in a data and/or address register.

3. **Address Register Indirect** - The operand is in a memory location pointed to by an address register.

4. **Indirect with Indexing** - The operand is in a memory location which is pointed to by the sum of an address register and an index register.

5. **Memory Direct (or Absolute)** - The operand is in a memory location which is supplied by an expression.

6. **Program Relative** - The operand is in a memory location which is offset from the PC by a given displacement.

7. **Immediate** - The operand follows the instruction word.

Implicit Addressing

Some instructions make implicit reference to the program counter (PC), the system stack pointer (SP), the supervisor stack pointer (SSP), the user stack pointer (USP), or the status register (SR). The table below provides a list of these instructions and the registers implied.

INSTRUCTION	IMPLIED REGISTER(S)
branch conditional (Bcc), branch (BR)	PC
break (BRK)	SSP SR
break on overflow (BRKV)	SSP SR
call to subroutine (CALL)	PC SP
check register against bounds (CHK)	SSP SR
test condition decrement and branch (DBcc)	PC
signed divide (DIV)	SSP SR
unsigned divide (DIVU)	SSP SR
link and allocate (LINK)	SP
move condition codes (MOVE CCR)	SR
move status register (MOVE SR)	SR
move user stack pointer (MOVE USP)	USP
push effective address (PUSHA)	SP
return from exception (RETI)	PC SSP SR
return and restore condition codes (RTR)	PC SP SR
return from subroutine (RET)	PC SP
unlink (UNLK)	SP

Register Direct Modes

The operand is in one of the 68ØØØ registers. So, it could be in either AØ-A7, DØ-D7, CCR, SR, or USP (USP can only be used in the supervisor mode).

Data Register Direct: .Dn

The operand is in the specified data register. For example,

```
     CLRW      .DØ
```

clears the least significant word in data register DO.

The operand is a data register; length (l) is 1, 2, or 4 bytes.

```
31                            Ø
 ------------------------------
|        OPERAND (Dn)          |
 ------------------------------
```

Address Register Direct: .An

The operand is in the specified address register. For example,

```
     MOVL      .AØ, .Al
```

moves the long word contents of address register AØ to address register Al.

The operand is an address register; length (l) is 2 or 4 bytes (this addressing mode is valid for word and long operations only).

31	Ø
OPERAND (An)	

Memory Address Modes

The operand is a memory location pointed to by the specific mode.

Address Register Indirect: @An

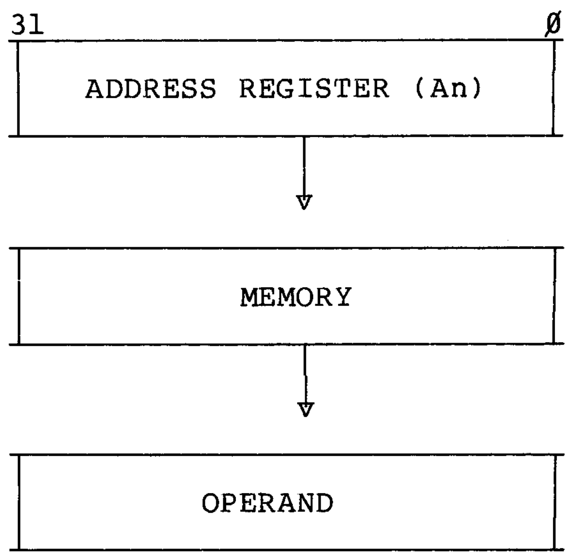
The operand is in a memory location pointed to by an address register. The addressing mode is used to address data.

For example:

```
     CRLB      @AØ
```

clears the byte at the memory location referenced by address register AØ.

The operand is in a memory location; length is 1, 2, or 4 bytes.



Address Register Indirect Postincrement: @An+

The operand is a memory location pointed to by the contents of an address register (AO-A7).

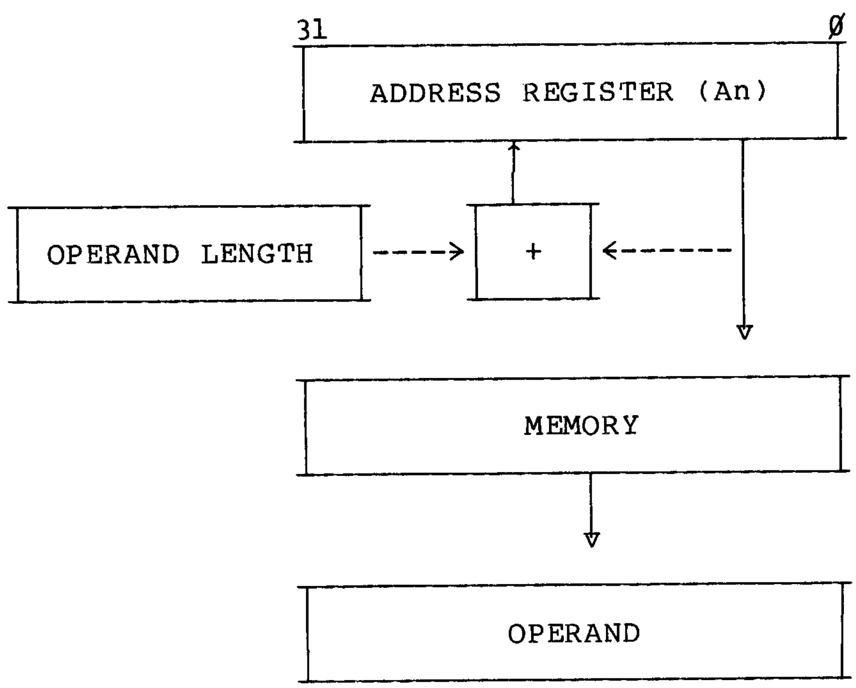
When the operation is complete, the address register is incremented by one, two, or four. The increment size depends on whether the size of the operand is byte, word, or long word. If the address register is the stack pointer (SP or A7) and the operand is one byte, the address is incremented by two rather than one to keep the stack pointer on a word boundary. This addressing mode causes data mode memory accesses.

For example:

```
     CLRW       @AØ+
```

clears the word at the memory location referenced by address register AØ and then increments AØ by 2.

The operand is in a memory location; length is 1, 2, or 4 bytes.



Address Register Indirect Predecrement: -@An

The operand is in a memory location pointed to by the contents of an address register (AO-A7).

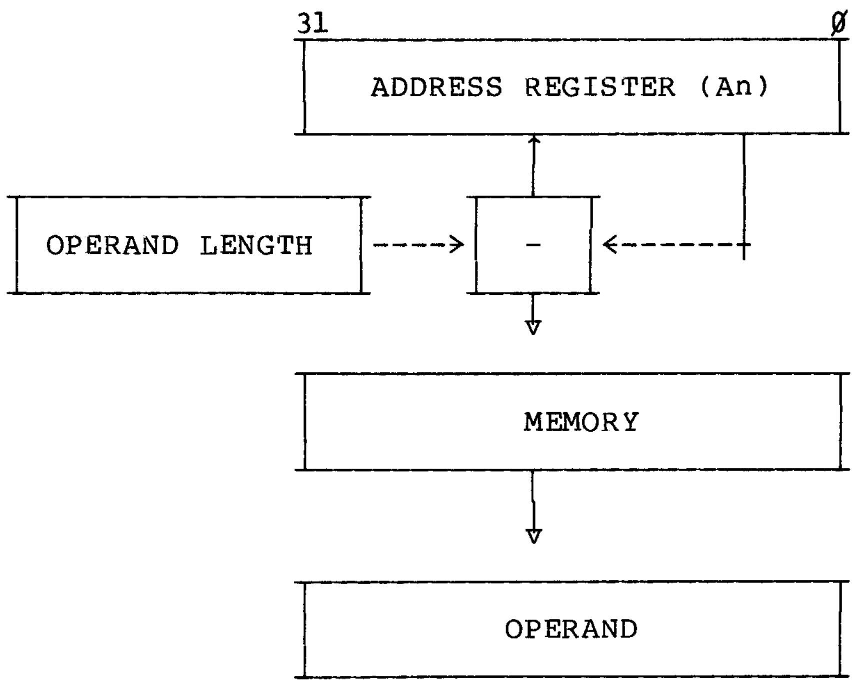
Before the operand location is used it is decremented one, two, or four. The decrement size depends on whether the operand is size is byte, word, or long word. If the address register is the stack pointer and the operand size is byte, the address is decremented by two rather than by one to keep the stack pointer on a word boundary. This addressing mode causes data mode memory accesses.

For example,

```
     CLRL      -@AØ
```

Decrements address register AØ by 4 and then clears the long word at the memory location referenced by AØ.

The operand is in a memory location; length is 1, 2, or 4 bytes.



Address Register Indirect
with 16-bit Displacement: /exp@An

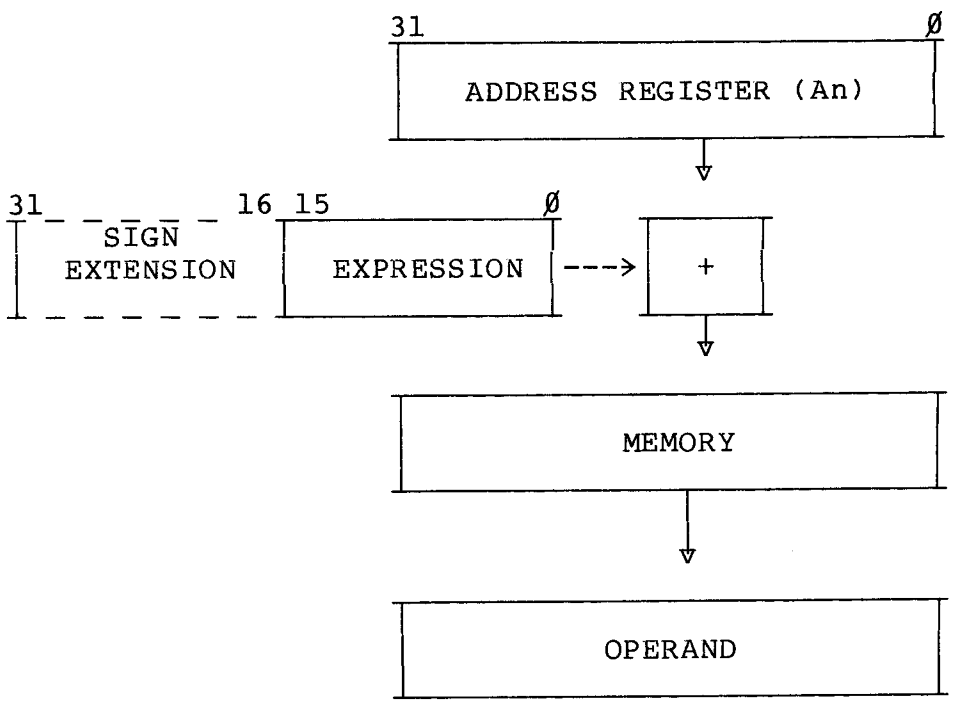
The operand is a memory location pointed to by the contents of an address register plus a 16 bit expression which is sign extended to 32 bits by the processor. This addressing mode causes data made memory addresses. By branch instruction, it can refer to program locations.

For example,

```
     CLR        8@AØ
```

clears the word at the memory location given by the sum of address register AØ and 8.

The operand is a memory location; length is (l) = 1, 2, or 4 bytes



Address Register Indirect Indexed with 8-bit Displacement: /exp@An(Ri)

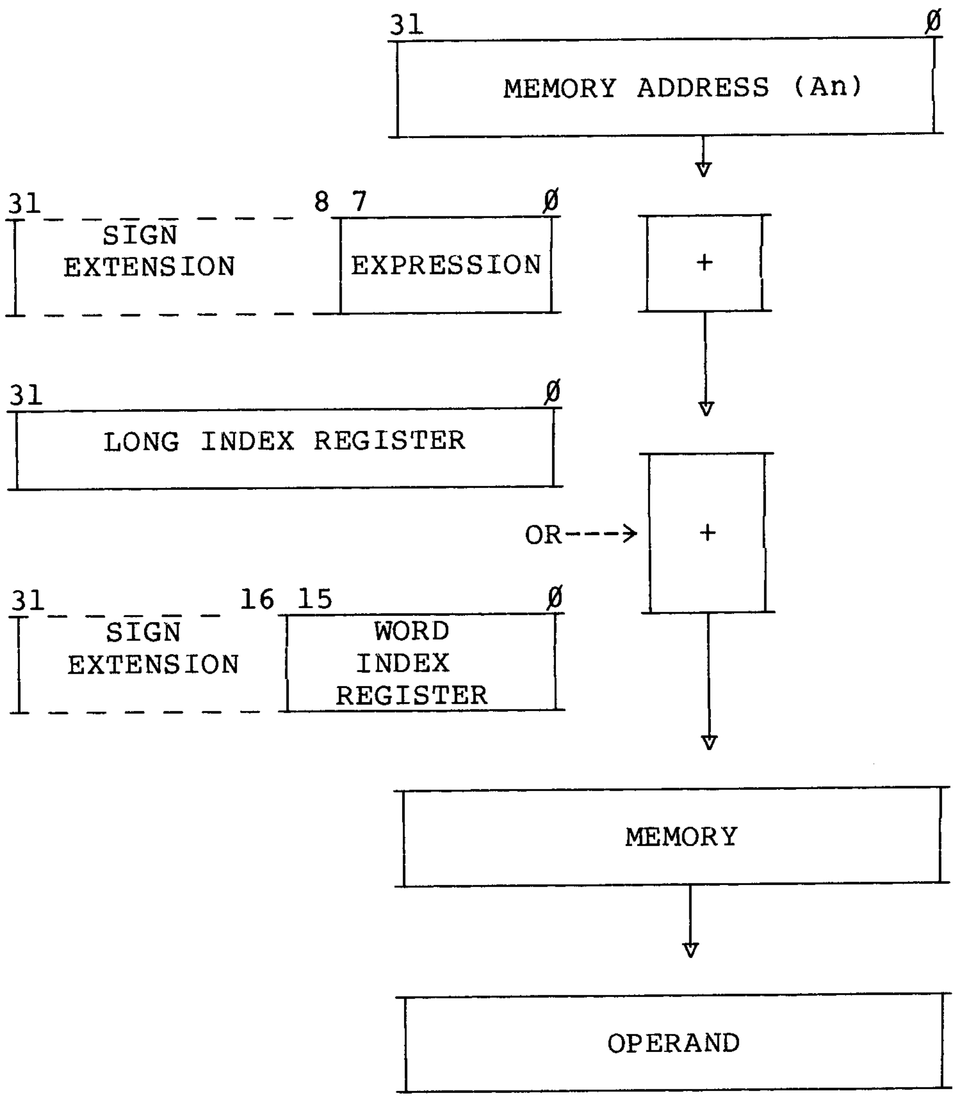
The operand is a memory location determined by the sum of the contents of the specified address register, an index register (either a double word or a sign-extended word) and the 8-bit expression which is sign extended to 32-bits by the processor. This mode causes data mode memory accesses, except when used with the BR and CALL instructions.

For example,

```
     CLRW      5@AØ(DØ)
```

clears the word at the memory location given by a sum of five, address register AØ, and index register DØ.

The operand is in a memory location; length is 1, 2, or 4 bytes.


31
Ø
MEMORY ADDRESS (An)
31
8 7
Ø
SIGN
EXTENSION
EXPRESSION
+
31
Ø
LONG INDEX REGISTER
OR--->
+
31
16 15
Ø
SIGN
EXTENSION
WORD
INDEX
REGISTER
MEMORY
OPERAND

Special Address Modes

The operand is a memory location pointed to by an expression.

Short Absolute: /exp [.W]

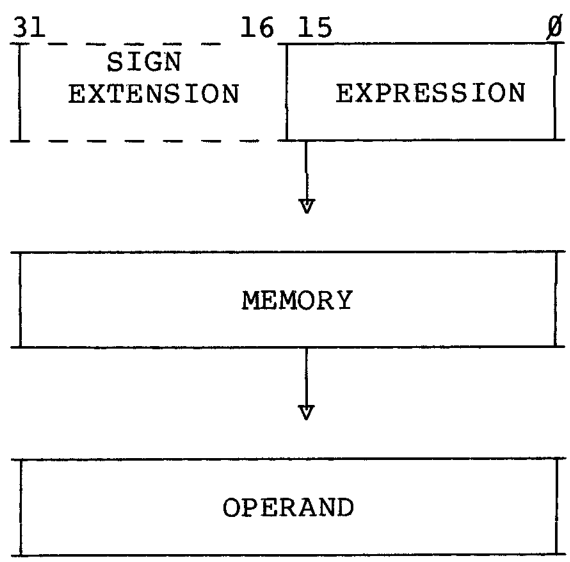
The operand is a memory location pointed to by a 16 bit expression which is sign-extended to 32 bits by the processor. This addressing mode causes data mode memory accesses. With branch instructions, it can be used to reference program locations.

For example,

```
     CLRW      /SUM.W
```

clears the word at the memory location given by SUM (the address of SUM has been assigned by a directive such as RES or DATA).

The operand is in a memory location; length is 1, 2, or 4 bytes.



Long Absolute: /exp[.L]

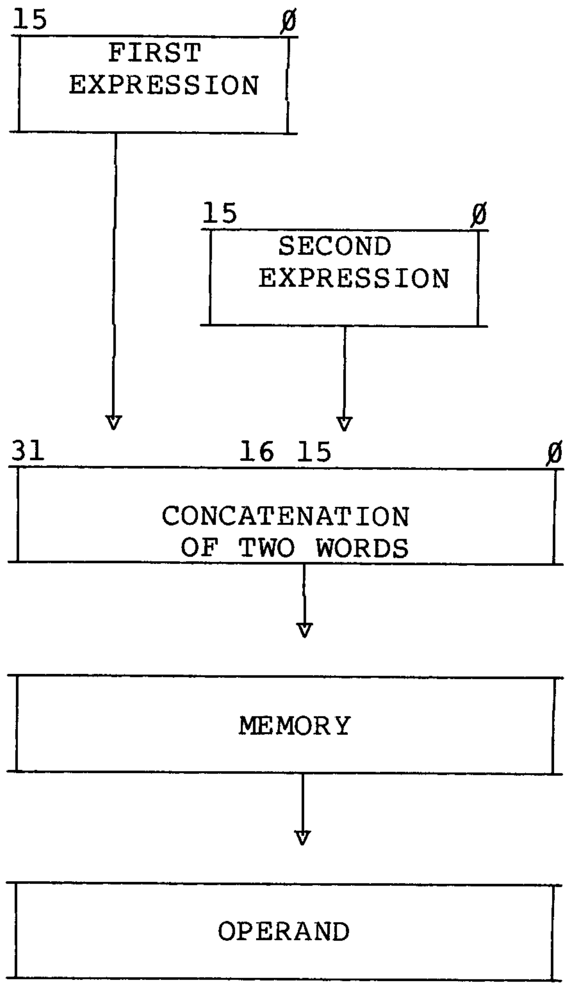
The operand is in a memory location pointed to by a 32 bit expression. The first word of the expression the high order part of the address. The second word of the expression is the low order part of the address. This addressing mode can be used to address data. With branch instructions it can be used to reference program locations.

For example,

```
     CLRW      /HEADING.L
```

clears the word at the memory location given by HEADING (the address of HEADING has been assigned by a directive such as RES or DATA).

The operand is in a memory location; length is 1, 2, or 4.



Program Relative: exp@PC

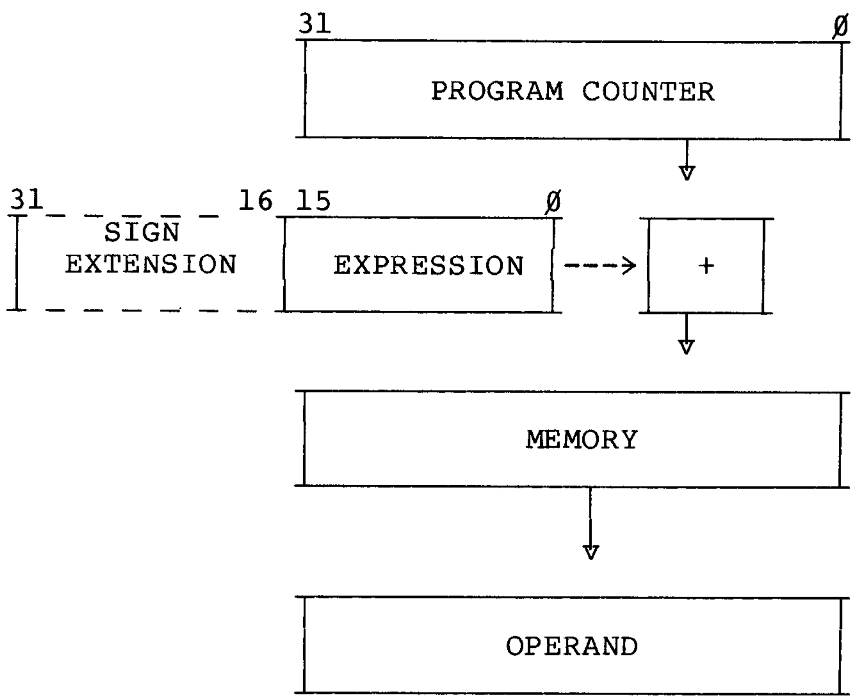
The operand is in a memory location pointed to by the sum of the program counter (PC) and a sign extended 16 bit expression. The value of the program counter is the address of the 16 bit displacement. This addressing mode causes program mode memory accesses. With the branch instruction it can be used to refer to program locations.

For example,

```
BR          LOOP
```

branches to memory location LOOP. Note that the "@PC" is optional here.

The operand is in a memory location; length is 1, 2, or 4 bytes.



Program Relative with Index: exp@PC(Ri)

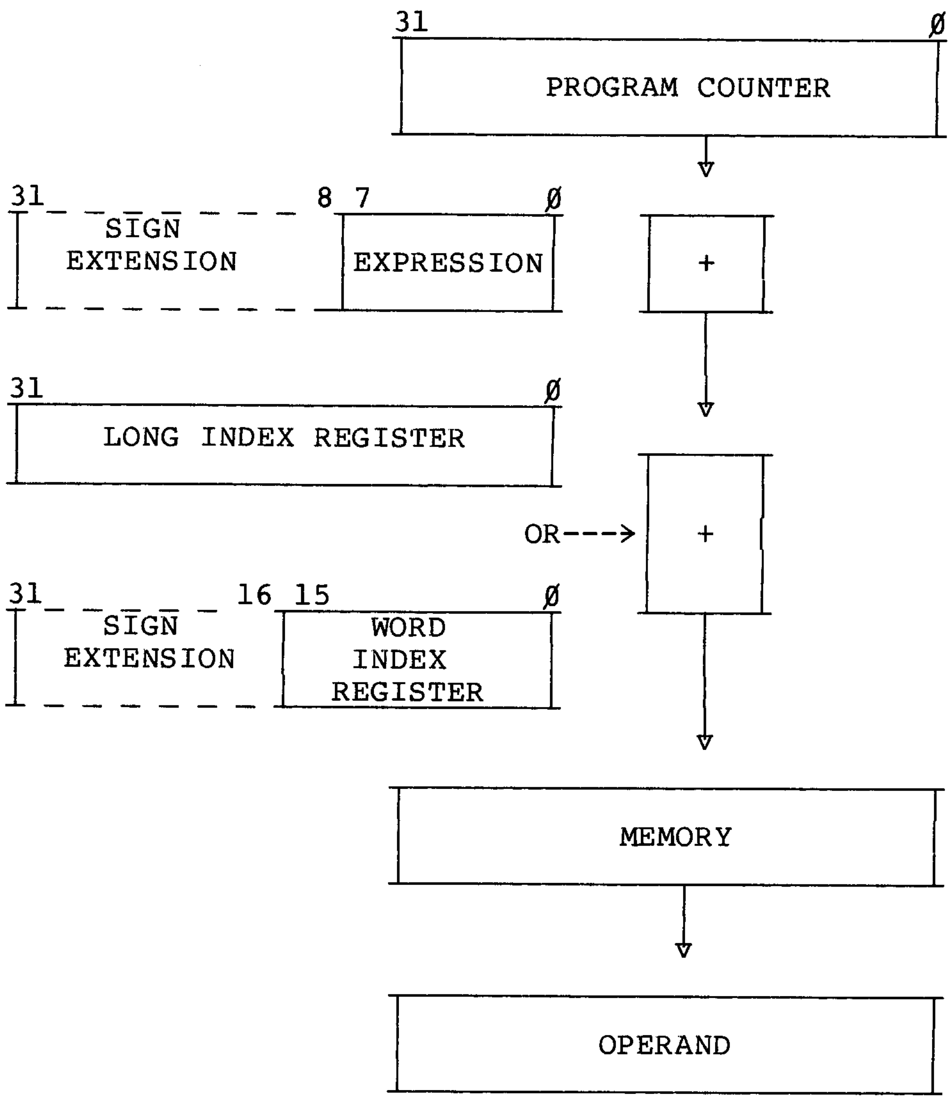
The operand is in a memory location pointed to by the sum of the program counter, the index register (Ri) and the 8-bit expression. Both the index register and the expression are sign-extended to 32 bits by the processor. The value in the program counter is the address of the displacement value. This addressing mode causes program mode memory references.

For example,

```
     CLRW       NUM@PC(D1)
```

clears the word at the memory location given by the sum of NUM, the PC, and data register D1.

The operand is in a memory location; length is 1, 2, or 4 bytes.



Immediate Data: #exp[.W]   #exp[.L]

The operand follows the instruction word in either one or two extension words, depending on the size of the operation:

Byte Operation - The operand is the low order byte of the first extension word.

Word Operation - The operand is the extension word.

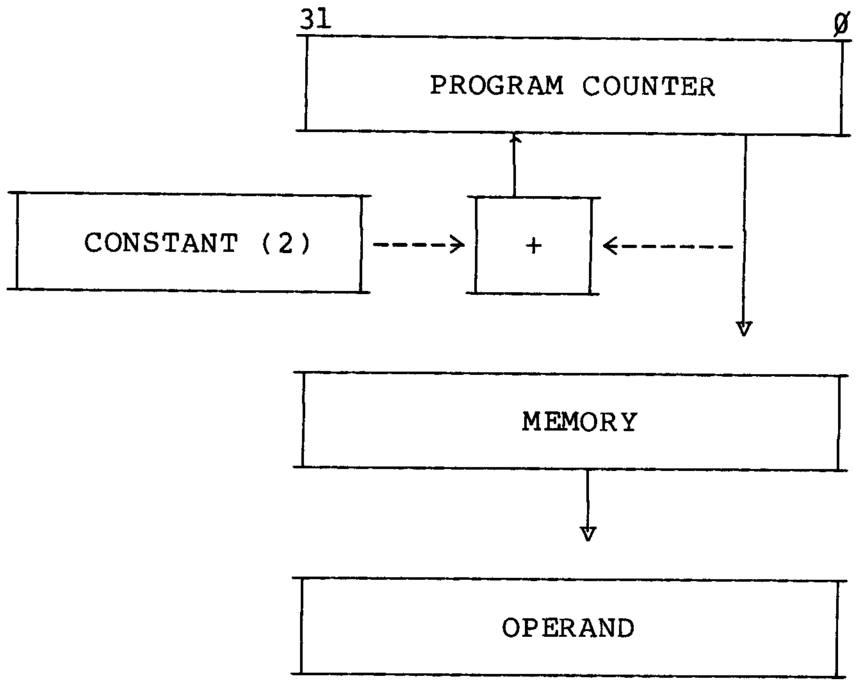
Long Operation - The operand is in the two extension words. The high order 16 bits are in the first extension word; the low order 16 bits are in the second extension word.

For example,

```
     LDW       .DØ,#H'112F
```

loads the data register DØ with the immediate data H'112F.

The operand is in the memory location immediately following the instruction; length is 1 or 2 bytes for 1 extension word and 4 bytes for 2 extension words.


31
0
PROGRAM COUNTER
CONSTANT (2)
+
MEMORY
OPERAND

CHAPTER 7

THE ASSEMBLER-16 PROGRAM

CHAPTER 7/ THE ASSEMBLER 16 PROGRAM

An assembly language source program can contain:

- Labels
- Instructions, Directives, or Programmed Operations
- Operands
- Comments

These are organized into a program line in one of the following ways:

[**label**]bb[**INSTRUCTION**]bb[**operand(s)** [1]]bb[**comment**],

[**label**]bb[**DIRECTIVE**]bb[**operand(s)** [1]]bb[**comment**] or

[**label**]bb[**PROGRAMMED OPERATION**]bb[**operand(s)** [1]]bb[**comment**]

All fields are described as optional. A statement will have at least one field. Others will be optional depending on the statement.

LABELS

A label statement begins in column 1. It can be either global or local. A label must be followed by at least two spaces.

Global -

If the label is global (accessible by any main programs or subroutines), it can contain up to 45 characters. The first character must be alphabetic, the next 45 nonblank characters can be alphabetic, numeric, or an underline. No more than one consecutive blank space is permitted in a symbol. Single blanks are not significant. Two global labels which match the first 45 nonblank characters are the same to the Assembler-16.

A global label can be the only entry on a source input line (this is known as a "hanging label"). Any reference to a hanging label will, in effect, be a reference to the statement following the hanging label.

Local -

If the label is a local label (defined only within the current program), it is defined on the current location counter. Local labels consist of a dollar-sign character ($) followed by a single integer. Local labels are used in the same way global labels are except that the scope is delimited by global labels.

INSTRUCTIONS

There are three types of instructions you can use:

- Mnemonics
- Directives
- Programed Operations

Mnemonics

The instruction (mnemonic) field can begin anywhere except in column one. If there is a label, two blanks must separate the label and instruction. The mnemonic field contains one of the allowed M68ØØØ operation mnemonics, followed by an operand length (l) indicator where needed.

Length -

With most instructions, you may specify the length of the operand on which it will act. This length (l) can be:

B - byte (8 bits)
W - word (16 bits)
L - long word (32 bits)

For example:

ADDB

signifies an ADD instruction of one byte (8 bits).

If you do not specify a length, the Assembler-16 assumes a length. If the Assembler-16 cannot determine the length, it assumes word length and issues a warning to that effect.

The following table illustrates the length assumed by the Assembler-16:

First Operand:		B	W	L	U
Second Operand:	B	B	B	B	B
	W	B	W	W	W
	L	B	W	L	L
	U	B	W	L	U

You cannot specify length on unsized instructions or on instructions with only one size possibility.

Directives

The same basics that apply the instructions, apply to directives:

The label if present, begins in column one with two blanks separating it from the directive. If there is no label, the directive can begin in column two. Some directives require either a label in the statement or a hanging label preceding the statement.

The operand field syntax depends on the directive. The field begins two or more blank spaces beyond the end of the directive field. The field is terminated by two consecutive blanks not inside a quoted string.

The optional comments' field is two blank spaces after the operand field. If there is no operand, it begins two positions after the directives field.

Programed Operations

The format for programmed operations is similar to that of directives:

The programmed operation field contains a predefined program operator. It is defined in a user-defined opcode directive statement (in the source program) prior to its use as an opcode. This is done with the FORM directive (see Chapter 9).

The operands are likewise predefined.

The label, if any, is defined at the current location counter.

OPERANDS

The optional operand field is separated from the mnemonic field by at least two blank spaces. The format of the operands depends upon the instruction; Ø, 1, 2, or more operands may be permitted. In certain addressing modes an operand length (l) can be optionally stipulated:

.W = word
.L = long word

General Operand Rules:

1. Multiple operands are separated by a comma.
2. Operands are checked by the Assembler-16 both by number and by addressing modes.
3. Each operand must be valid for the instruction being performed.
4. Single blank spaces are permitted within symbols and around operators and special characters.
5. Two or more blank spaces terminate the operand field and begin the comment field unless the blank spaces are within a quoted string.

COMMENTS

The optional comment field is separated from the previous field by at least two blanks. You cannot have a comment with a hanging label, nor may you have a comment on an END statement which does not have an entry point specified. A comment can occupy a whole line if there is an asterisk (*) in column one.

EXPRESSIONS

Expressions occur as operands of machine instruction, assembler directive statements, or as programmed operation statements. An expression consists of one or more terms separated by optional operators.

Each term in an expression may be:

- a self-defining constant,
- a symbol (local or global),
- the program location counter character (*), or
- a parenthesized expression.

Constants may be decimal, hexadecimal, binary, octal, or character constants:

> Hexadecimal constants consist of a string of hex digits preceded by capital H quote (H').
>
> Binary constants consist of a string of binary digits preceded by capital B quote (B').
>
> Octal constants consist of a string of octal digits preceded by capital Q quote (Q').
>
> A Character constant is a single character preceded by a quote (single or double). Packed character constants are not allowed.

A **local or global symbol** represents an address. The Assembler-16 uses the symbol as a displacement to the PC register.

The **program counter** (PC) results in a 32 bit displacement of the current statement.

Expression evaluation is left to right with unary operator precedence. Parentheses may be used to change the order of expression evaluation. The operators used to build an expression are arithmetic or logical.

Arithmetic operators act upon 32 bit signed integer quantities with negative numbers in twos complement format.

> \+ is addition

```
-  is unary minus or subtraction
*  is multiplication
/  is signed division
```

Only addition and subtraction are permitted using relocatable values.

Logical operators act upon 32 bit binary unsigned numbers.

```
.AND. is logical AND
.XOR. is logical XOR
.OR.  is logical OR
.NOT. is logical NOT
.SHL. is shift left logical
```

where

```
A.SHL.B
```

shifts value A left B bits if B is positive and shifts value A right (-B) bits if B is negative.

The Assembler-16 computes the size attribute of an expression from the size attribute of the terms of the expression. If exactly one term has a non-U size attribute, and all other terms have a U size attribute, then the expression inherits the non-U size attribute. If more than one term has a non-U size attribute, or if all terms have a U size attribute, then the expression is assigned a size attribute of U.

CHAPTER 8

INSTRUCTIONS

CHAPTER 8/ INSTRUCTIONS

Description

In this chapter, each instruction is listed alphabetically. They are either listed individually or as part of an instruction group.

Instruction Groups

Some of the instructions fall into an instruction group. Each instruction within the group uses the same mnemonic, but the assembler-16 translates them into different machine codes. Preceding each instruction group is an overview giving the general function of the instruction group.

Individual Instructions

Some instructions have only one form. The function of individual instructions follows its syntax.

Syntax

The syntax consists of the mnemonic, the sizes allowed, and the addressing modes permitted. For example:

```
LDA   .Ad,/exp[.W or .L]       Operand length(l) = L
            .Ad,[exp]@As[(Ri)]
            .Ad,exp[@PC[(Ri)]]
```

The operand length is long (L); hence the mnemonic is LDA. The destination operand is an address register (.Ad), and the source operand may be either :

1. an absolute short or long address ("/exp[.W or .L]")
2. an indirectly derived address with optional displacement and index ("[exp]@As[(Ri)]")
3. a program relative address with optional index ("exp[@PC[(Ri)]]")

Condition Codes

After execution of the instruction, the status of the condition codes may change. This change is reflected under "Condition Codes". Each bit of the condition code register (CCR) may be listed as:

- \- the bit is unaffected by the instruction
- Ø the bit is cleared (reset to Ø) after the instruction is executed
- * the bit is either set to 1 or reset to Ø based on the result of the operation
- U the bit is undefined for the operation

For example:

X	N	Z	V	C
-	*	*	Ø	Ø

This means that the X bit is not affected, the N and Z bits are set or cleared based on the result of the operation, and the V and C bits are always cleared.

Instruction Fields

The Assembler-16 translates the instruction into machine code. This machine code is referred to as the "Instruction Field". For example:

15	14	13	12	11	1Ø	9	8	7	6	5	4	3	2	1	Ø
Ø	1	Ø	Ø	Ø	Ø	1	Ø	size		mode			reg.		

This is the instruction code for CLeaR. The top line refers to the bit number in the instruction word and the numbers in the second line refer to the actual bit. In this example, bits 8-15 contain the machine code for the CLeaR instruction.

The word "size" is listed under bits 6-7. This refers to the size of the operand to be CLeaRed (i.e.,byte, word, or long word). Under bits 3-5 is the word "mode". This refers to the addressing mode (e.g. direct address register, short address, etc.) The addressing mode takes three bits to describe, and these codes are listed on the next few pages.

Under bits Ø-2 is the word "register". This refers to the register number used by the instruction. This value also requires three bits and the codes for each register are listed on the following pages. For example, the instruction:

```
        CLRL      .D1
```

generates the following code:

15	14	13	12	11	1Ø	9	8	7	6	5	4	3	2	1	Ø
Ø	1	Ø	Ø	Ø	Ø	1	Ø	1	Ø	Ø	Ø	Ø	Ø	Ø	1

(Note: The instruction codes may vary from the example above, depending on the number of operands, the sizes allowed, etc.)

REGISTER/MODE CODES

Function Code	Reg./mode Mnemonic	Mode Code	Reg.
Data Register Direct	= .Dn	ØØØ	n
Address Register	= .An	ØØ1	n
Address Register Indirect	= @An	Ø1Ø	n
Address Register Indirect Postincrement	= @An+	Ø11	n
Address Register Indirect Predecrement	= -@An	1ØØ	n
Address Register Indirect 16-bit displacement	= exp@An	1Ø1	n
Address Register Indirect Indexed with 8 bit displacement.	= exp@An(Ri)	11Ø	n

This mode requires an additional one word extension:

15	14 13 12	11	1Ø	9	8	7 6 5 4 3 2 1 Ø
D/A	reg.	W/L	Ø	Ø	Ø	displacement value

Bit 15 -
    Ø = data register is index register
    1 = address register is index register

Bits 14 through 12 -
    index register number

Bit 11-
    Ø = sign extended, low order integer in index register
    1 = long value in index register

Absolute Short Address	= /exp[.W]	111	ØØØ
Absolute Long Address	= /exp[.L]	111	ØØ1
Program Relative	= exp @PC	111	Ø1Ø
Program Relative with Index	= exp @PC(Ri)	111	Ø11

This address mode requires one word of extension:

15	14 13 12	11	1Ø	9	8	7 6 5 4 3 2 1 Ø
D/A	register	W/L	Ø	Ø	Ø	displacement value

Bit 15 -
    Ø = data register
    1 = address register

Bits 14 through 12 -
    index register number

Bit 11 - Index size
    Ø = sign extended, low order word in index register.
    1 = long value in Index Register.

Immediate Short	= #exp[.W]	111	1ØØ
Immediate Long	= #exp[.L]	111	1ØØ
Status Register	= .SR	111	1ØØ
Condition Code Register	= .CCR	111	1ØØ

ADD
ADD binary

This instruction's general operation is:

ADD[l] destination, source

which adds the source to the destination. The result is stored in the destination.

The Assembler-16 can interpret the binary ADD four different ways. From the operands used, it determines which instruction is to be executed.

The Assembler-16 chooses which operation to use according to the following guidelines:

ADD quick	if the source is immediate (indicated by the # sign) and less than or equal to 8.
ADD address	if the destination is an address register and the source is other than immediate.
ADD immediate	if the source is immediate and greater than 8 (more than 3 bits).
ADD data register	for any other operations performed by the ADD instruction. A data register is always one of the operands.

ADD
ADD quick/ ADD immediate

```
ADD[l]        .Dd, #exp[.W or .L]  Operand length(l): B W L
             -@Ad, #exp[.W or .L]
             @Ad+, #exp[.W or .L]
/expl[.W or .L], #exp2[.W or .L]
[expl]@Ax[(Ri)], #exp2[.W or .L]
              .Ad, #exp[.W or .L] (Quick only (l)=W L only)
```

Condition Codes:

X	N	Z	V	C
*	*	*	*	*

X - Set if carry occurred, cleared otherwise (same as carry(C)).
N - Set if result is negative, cleared otherwise.
Z - Set if result is zero, cleared otherwise.
V - Set if overflow generated, cleared otherwise.
C - Set if carry occurred, cleared otherwise.

Condition codes are not affected if an addition to an address register is made.

Examples:

If DØ is H'8Ø2A, then

```
ADDW    .DØ,#H'7ØØØ
```

changes the contents of DØ to H'FØ2A

If DØ contains H'ØØ2A, then

```
ADDW    .DØ,#4
```

Changes the contents of DØ to H'ØØ2E.

Instruction Fields for ADD quick

15	14	13	12	11	1Ø	9	8	7	6	5	4	3	2	1	Ø
Ø	1	Ø	1	data			Ø	size		mode			reg.		

The data field contains bit data, with values 1-8.
    (ØØ1-111 = 1-7 decimal ØØØ = 8 decimal)

The size field contains the size of the operation.
    If the size field contains ØØ it is a byte operation.
    If the size field contains Ø1 it is a word operation.
    If the size field contains 1Ø it is a long word operation.

The mode and register fields contain the address mode of the destination operand. Address register direct addressing is not permitted when the size of the instruction is byte length. If the size is word length and the destination is an address register, the source is sign extended to 32-bits.

Instruction Fields for ADD immediate

According to the size of the operation, the number of extensions for the immediate data vary (see data fields).

15	14	13	12	11	1Ø	9	8	7 6	5 4 3	2 1 Ø
Ø	Ø	Ø	Ø	Ø	1	1	Ø	size	mode	reg.

+

	byte data (8 bits)

or

word data (16 bits)

or

long word (32 bits including previous word)

The size field contains the size of the operation.
    If the size field contains ØØ it is a byte operation.
    If the size field contains Ø1 it is a word operation.
    If the size field contains 1Ø it is a long word operation.

The mode and register fields contain the address mode of destination operand.

The data field contains the data immediately following the instruction:

If the size field contains ØØ, then the data is in the low order byte of the immediate word (8 bits).
If the size field contains Ø1, then the data is in the entire immediate word (16 bits).
If the size field contains 1Ø, then the data is in the next two immediate words (32 bits).

ADD
ADD address register

```
ADD[l]   .Ad, .As                       Operand Length(l):W,L
         .Ad, .Ds
         .Ad, -@As
         .Ad, @As+
         .Ad, /exp[.W or .L]
         .Ad, [exp]@Ay[(Ri)]
         .Ad, exp[@PC[(Ri)]]
         .Ad, #exp[.W or .L] (Add immediate only)
```

Adds the source to the destination address register.

Example:

If AØ contains H'6ØØØ and Al contains H'ØØlØ, then

```
    ADDW    .AØ,.Al
```

changes the contents of AØ to H'6Ø1Ø.

Condition Codes:

X	N	Z	V	C
-	-	-	-	-

All of the flags are unaffected.

Instruction Fields:

15	14	13	12	11	1Ø	9	8	7	6	5	4	3	2	1	Ø
1	1	Ø	1	register			size			mode			reg.		

The register field contains any address register; always the destination operand.

The size field contains the size of the operand.

> If the size field contains ØØ1 it is a word operation. The source operand is sign-extended (See EXT) to fill 32 bits of the address register.

If the size field contains 111 it is a long word operation.

The mode and register fields contain the address mode of the source operand.

ADD
ADD data register

Operand length(l):B,W,L

```
ADD[l]  .Dd, .Ds
        .Dd, -@As
        .Dd, @As+
        .Dd, /exp[.W or .L]
        .Dd, [exp]@Ay[(Ri)]
        .Dd, exp[@PC[(Ri)]]
        .Dd, .As      (Word and long word length only)
        -@Ad,.Ds
        @Ad+,.Ds
        /exp[.W or .L],.Ds
        [exp]@Ax[(Ri)],.Ds
```

A data register is always one of the operands.

Example:

If DØ contains H'Ø143 and Dl contains H'ØØØ7, then

```
ADDW     .DØ,.Dl
```

changes the contents of DØ to H'Ø14A.

Condition Codes:

X	N	Z	V	C
*	*	*	*	*

X - Set if carry occurred, cleared otherwise(same as carry(C)).
N - Set if the result is negative, cleared otherwise.
Z - Set if the result is zero, cleared otherwise.
V - Set if overflow is generated, cleared otherwise.
C - Set if carry occurred, cleared otherwise.

Instruction Fields:

15	14	13	12	11 10 9	8 7 6	5 4 3	2 1 Ø
1	1	Ø	1	register	size/op	mode	reg.

The register field contains the data register.

The size/op field contains the size of the operation and the destination of the result.

Byte	Word	Long Word	Destination
ØØØ	ØØ1	Ø1Ø	data register
1ØØ	1Ø1	11Ø	second operand

The mode and register fields contain the location of the second operand.

> If the second operand is the source operand and its size is one byte, the address register direct addressing mode is not permitted.

ADDC
ADD binary with Carry

```
ADDC[l] .Dd,.Ds                    Operand length (l): B,W,L
        -@Ad,  -@As
```

Adds the source and carry bit at the destination. The result is stored in the destination.

Example:

If DØ contains H'Ø5 and D1 contains H'Ø6, and the extend bit of the status register is set, then

```
ADDCB    .DØ,.D1
```

changes the contents of DØ to H'ØC.

Condition Codes:

X	N	Z	V	C
*	*	*	*	*

X - Set if carry is generated, cleared otherwise.
N - Set if the result is negative, cleared otherwise.
Z - Cleared if the result is nonzero, unchanged otherwise.
V - Set if overflow is generated, cleared otherwise.
C - Set if carry is generated, cleared otherwise.

Instruction Fields:

15	14	13	12	11	1Ø	9	8	7	6	5	4	3	2	1	Ø
1	1	Ø	1	reg(d)			1	size		Ø	Ø	r/m	reg(s)		

The register(d) field contains the destination register.

If the R/M field is zero, register(d) is the data register.
If the R/M field is one, register(d) is the address register in the predecrement addressing mode.

If the size field contains Ø1 it is a word operation.
If the size field contains 1Ø it is a long operation.

The R/M field contains the operand addressing mode.
If the R/M field is Ø, the operation is from data register to data register.
If the R/M field is 1, the operation is from memory to memory.

The register(s) field contains the source register.
If the R/M field is Ø, register(s) is the data register.
If the R/M field is 1, register(s) is the address register in predecrement addressing mode.

ADDD
ADD Decimal(BCD) with extend

ADDD .Dd, .Ds
-@Ad, -@As

Operand length (1): B

Adds the source and the extend bit to the destination and then stores it in the destination. Binary coded decimal arithmetic is used for the addition.

Example:

If DØ contains 6 and D1 contains 7, and the extend bit of the status register is set, then

```
ADDD     .DØ,.D1
```

changes the contents of DØ to 14.

Condition Codes:

X	N	Z	V	C
*	U	*	U	*

X - Set if a carry (BCD) is generated, cleared otherwise.
N - Undefined.
Z - Cleared if the result is non-zero. Unchanged otherwise.
U - Undefined.
C - Set if a carry (decimal) generated. Cleared otherwise.

Instruction Fields:

15	14	13	12	11	1Ø	9	8	7	6	5	4	3	2	1	Ø
1	1	Ø	Ø	reg(d)			1	Ø	Ø	Ø	Ø	R/M	reg(s)		

If the R/M field is Ø, the operation is from data register to data register.
If the R/M field is 1, the operation is from memory to memory.

The register(s) field contains the source register.

The register(d) field contains the destination register.

AND
logical AND

The AND instruction can be interpreted by the Assembler-16 two different ways. From the operands used, the Assembler-16 determines which instruction to use.

AND[1] destination, source

which ANDs the source with the destination. The result is stored in the destination.

The Assembler-16 chooses which operation to use according to the following guidelines:

AND immediate	if the source is immediate (indicated by the # sign).
AND data	for any other operations performed by the AND instruction. A data register is always one of the operands.

Condition Codes: (Identical for both operations)

X	N	Z	V	C
-	*	*	Ø	Ø

X - Not affected
N - Set if the most significant bit of the result is set, cleared otherwise.
Z - Set if the result is zero, cleared otherwise.
V - Always cleared.
C - Always cleared.

AND
logical AND immediate

```
AND[l]          .Dd, #exp[.W or .L]    Operand length(l): B,W,L

              -@Ax, #exp[.W or .L]
              @Ax+, #exp[.W or .L]
/expl[.W or .L], #exp2[.W or .L]
[expl]@Ax[(Ri)], #exp2[.W or .L]
              .CCR, #exp[.W or .L] (l=W only)
```

Example:

If DØ contains H'4F, then

```
    ANDB      .DØ,#H'FØ
```

changes the contents of DØ to H'4Ø.

Instruction Fields:

15	14	13	12	11	1Ø	9	8	7	6	5	4	3	2	1	Ø
Ø	Ø	Ø	Ø	Ø	Ø	1	Ø	size		mode			reg.		

+

	byte data (8 bits)

or

word (16 bits)

or

long word (32 bits)

The size field contains the size of the operation.
 If the size field contains ØØ it is a byte operation.
 If the size field contains Ø1 it is a word operation.
 If the size field contains 1Ø it is a long word operation.

The register and mode fields contain the address mode of the destination operand

The data field contains the data immediately following the instruction:

If the size field contains ØØ, the data is in the low order byte of the immediate word (8 bits).
If the size field contains Ø1, the data is the entire immediate word (16 bits).
If the size field contains 1Ø, the data is the next two immediate words (32 bits).

AND
logical AND data

```
AND[l]  .Dd, .Ds                     Operand length(l):B,W,L
        .Dd, -@Ay
        .Dd, @Ay+
        .Dd, /exp[.W or .L]
        .Dd, [exp]@Ay[(Ri)]
        .Dd, exp[@PC[(Ri)]]
        -@An,.Ds
        @Ax+,.Ds
        /exp[.W or .L],.Ds
        [exp]@Ax[(Ri)],.Ds
```

Example:

If DØ contains H'FFØØ FFØØ and Dl contains H'ØØFF ØØFF, then

```
ANDL     .DO,.Dl
```

changes the contents of DØ to H'ØØØØ ØØØØ.

Instruction Fields:

15	14	13	12	11	1Ø	9	8	7	6	5	4	3	2	1	Ø
1	1	Ø	Ø	register			size/op			mode			reg.		

The register field contains the data register.

The size/op field contains the size of the operation and the destination of the result:

Byte	Word	Long Word	Destination
ØØØ	ØØ1	Ø1Ø	data register
1ØØ	1Ø1	11Ø	second operand

The register and mode fields contain the location of the second operand.

Bcc
Branch on condition

Bcc[l] exp[@PC] **Operand length(l): B,W**

Tests a condition. If the condition is true, sets the program counter (PC) to the value of the operand. If the condition is false, program execution continues at the next instruction.

Example:

If the zero flag of the status register is set, and LOOP is a statement label somewhere in the program, then,

```
     BE        LOOP
```

transfers control of the program to the instruction at LOOP.

Condition Codes:

X	N	Z	V	C
-	-	-	-	-

None of the flags are affected.

Instruction Fields:

15	14	13	12	11	1Ø	9	8	7	6	5	4	3	2	1	Ø
Ø	1	1	Ø	condition				8 bit displacement							
16 bit displacement (if 8 bit dislacement is Ø)															

The 8 bit displacement field contains the two's complement integer which contains the relative distance (in bytes) between the current instruction address, plus 2, and the referenced instruction.

The 16 bit displacement field allows larger displacement than 8-bits. It is used if the 8-bit field is equal to zero.

The condition field is one of the following fourteen conditions:

```
Ø111 BE  - equal
Ø11Ø BNE - not equal
Ø1Ø1 BC  - carry
Ø1ØØ BNC - no carry
1Ø1Ø BP  - positive
1Ø11 BN  - negative
1ØØ1 BV  - overflow
1ØØØ BNV - no overflow
111Ø BGT - greater than
11ØØ BGE - greater than or equal
11Ø1 BLT - less than
1111 BLE - less than or equal
ØØ1Ø BH  - higher than
ØØ11 BNH - not higher than
Ø1Ø1 BLO - low
Ø1ØØ BHS - high, same
```

BR
BRanch control addressing

BR /exp **Operand length(1): unsized**
[exp]@Ax[(Ri)]
exp[@PC(Ri)]

The program execution branches to the address given by the operand.

Example:

If LOOP is a statement label somewhere in the program, then

```
     BR      LOOP
```

transfers control of the program to the instruction at LOOP.

Condition Codes:

X	N	Z	V	C
-	-	-	-	-

None of the flags are affected.

Instruction Fields:

15	14	13	12	11	1Ø	9	8	7	6	5	4	3	2	1	Ø
Ø	1	Ø	Ø	1	1	1	Ø	1	1	mode			reg.		

The register and mode fields contain the address of the next instruction.

BRK
BReaK

BRK #exp Operand length(1): Unsized

Three operations are performed by this instruction:

1. The program counter is pushed into the supervisor stack.

2. The status register is pushed into the supervisor stack.

3. The vector number (#exp) is loaded into the program counter.

This instruction initiates exception processing. The vector number is generated to reference the Break exception vector which is specified by the four low order bits of the instruction. Sixteen vectors are available.

Example:

If a supervisory call (SVC) has been set up, then

```
BRK     #Ø
```

causes execution of that SVC.

Condition Codes:

X	N	Z	V	C
-	-	-	-	-

None of the flags are affected.

Instruction Fields:

15	14	13	12	11	1Ø	9	8	7	6	5	4	3	2	1	Ø
Ø	1	Ø	Ø	1	1	1	Ø	Ø	1	Ø	Ø	#exp			

The #exp field contains the number (break vector) to be loaded into the program counter.

BRKV
BReaK on oVerflow

BRKV Operand length(1): Unsized

Initiates exception processing if the overflow condition is on (overflow(V) = 1). Generates the vector number to reference the overflow exception vector as follows:

Program Counter (PC) --> Stack
Status Register (SR) --> Stack
Overflow Vector --> Program Counter (PC)

No operation is performed if the overflow is off. Execution continues with the next instruction. (Note: You must set the overflow vector using the SETTRP SVC before executing the exception.)

Example:

If the overflow bit of the status register is set, then

```
BRKV
```

initiates overflow exception processing.

Condition Codes:

X	N	Z	V	C
-	-	-	-	-

None of the flags are affected.

Instruction Fields:

15	14	13	12	11	1Ø	9	8	7	6	5	4	3	2	1	Ø
Ø	1	Ø	Ø	1	1	1	Ø	Ø	1	1	1	Ø	1	1	Ø

CALL
CALL general

```
CALL    /exp                      Operand length(1): Unsized
       [exp]@Ax[(Ri)]
        exp[@PC(Ri)]
```

Example:

If SUB1 is the label of a subroutine somewhere in the program, then

```
     CALL      SUB1
```

transfers control of the program to the instruction at SUB1.

Condition Codes:

X	N	Z	V	C
-	-	-	-	-

None of the flags are affected.

Instruction Fields:

15	14	13	12	11	1Ø	9	8	7	6	5	4	3	2	1	Ø
Ø	1	Ø	Ø	1	1	1	Ø	1	Ø	mode			reg.		

The register and mode fields contain the address of the next instruction.

CHK
CHecK against bounds

```
CHK                    .Du, .Dn        Operand Sizes : W
                      -@Au, .Dn
                      @Au+, .Dn
          /exp[.W or .L], .Dn
         [exp] @Ax[(Ri)], .Dn
           exp[@PC[(Ri)]], .Dn
            #exp[.W or .L], Dn
```

Examines the content of the low order in the data register (Dn) and compares it to the upper bound operand. The upper bound is a two's complement integer. Exception processing is initiated if the data register is less than zero or greater than the upper bound operand. Generates the vector number to reference the CHK instruction exception vector. (Note: You must set the CHK vector with the SETTRP SVC before executing this statement.)

Example:

If DØ contains H'Ø1ØØ and Dl contains H'Ø1Ø1, then

```
     CHK     .DØ,.Dl
```

initiates CHK exception processing.

Condition Codes:

X	N	Z	V	C
-	*	U	U	U

X - Not affected.
N - Set if Dn is less than zero; cleared if Dn is greater than the; undefined otherwise.
Z - Undefined.
V - Undefined.
C - Undefined.

Instruction Fields:

15	14	13	12	11	1Ø	9	8	7	6	5	4	3	2	1	Ø
Ø	1	Ø	Ø	data reg.			1	1	Ø	mode			reg.		
										(upper bound)					

The data register contains the data register whose content is checked.

The register and mode fields contain the upper bound operand word.

CLR
CLeaR an operand

```
CLR[l]  .Dn                    Operand length(l): B,W,L
        -@An
        @An+
        /exp[.W or .L]
        [exp]@An[(Ri)]
```

Clears the operand to all zero bits.

Example:

If DØ contains H'ØØØ1, then

```
CLRW    .DØ
```

changes the contents of DØ to H'ØØØØ.

Condition Codes:

X	N	Z	V	C
-	Ø	1	Ø	Ø

X = Not affected.
N = Always cleared.
Z = Always set.
V = Always cleared.
C = Always cleared.

Instruction Fields:

15	14	13	12	11	1Ø	9	8	7	6	5	4	3	2	1	Ø
Ø	1	Ø	Ø	Ø	Ø	1	Ø	size		mode			reg.		

The register and mode fields contain the operand addressing mode.

The size field contains the size of the operation:

If the size field contains ØØ it is a byte operation.
If the size field contains Ø1 it is a word operation.
If the size field contains 1Ø it is a long word

operation.

CMP
CoMPare

This instruction's general operation is:

CMP[1] Destination, Source

where the source is subtracted from the destination and the condition codes (CCR) are set according to the results. The values of the operands are not changed.

The compare instruction is interpreted by the Assembler-16 as four different instructions. The Assembler-16 determines which is to be executed by the operands.

The Assembler-16 chooses which operation to use according to the following guidelines:

CMP immediate	if the source is immediate (indicated by the # sign).
CMP memory	if both operands are addressed with the postincrement addressing mode.
CMP address	for any other compare operation where the destination is addressed using the address register direct mode.
CMP data	for any other compare operation where the destination is addressed using the data register direct mode.

Condition Codes: are identical for all CMP operations.

X	N	Z	V	C
–	*	*	*	*

X - Not affected.
N - Set if the result is negative, cleared otherwise.
Z - Set if th eresult is zero, cleared otherwise.

V - Set if overflow is generated, cleared otherwise.
C - Set if borrow is generated, cleared otherwise.

CMP
CoMPare immediate

```
CMP[l]      .Dd, #exp[.W or .L]     Operand length(l): B,W,L
            -@Ax, #exp[.W or .L]
            @Ax+, #exp[.W or .L]
            /expl[.W or .L], #exp2[.W or .L]
            [expl]@Ax[(Ri)], #exp2[.W or .L]
```

Example

If DØ contains H'ØØ1Ø, then

```
        CMPW        .DØ,#H'1Ø
```

sets the zero bit of the status register.

Instruction Fields:

15	14	13	12	11	1Ø	9	8	7	6	5	4	3	2	1	Ø
Ø	Ø	Ø	Ø	1	1	Ø	Ø	size		mode			reg.		

+

	byte data (8 bits)

or

word data (16 bits)

or

long word data (32 bits, including previous word)

The size field contains the size of the operation.
If the size field is ØØ, it is a byte operation.
If the size field is Ø1, it is a word operation.
If the size field is 1Ø, it is a long word operation.

The register and mode fields contain the address mode of the destination operand.

The data field contains the data immediately following the operation word.
If the size field is ØØ, the data is low order byte of the immediate word (8 bits).
If the size field is Ø1, the data is the entire

immediate word (16 bits).
If the size field is 1Ø, the data is the next two immediate words (32 bits).

CMP
CoMPare memory

CMP[l] @Ad+, @As+ Operand length(l): B,W,L

Example:

If AØ points to memory address H'4ØØØ, which contains H'ØØ, and Al points to address H'4ØØ4, which contains H'Ø5 then

```
     CMPB      @AØ+,@A1+
```

sets the negative and carry bits of the status register and increments AØ and Al by 1.

Instruction Fields:

15	14	13	12	11	1Ø	9	8	7	6	5	4	3	2	1	Ø
1	Ø	1	1	reg.(d)			1	size		Ø	Ø	1	reg.(s)		

The register(d) field contains the destination register.

The register(s) field contains the source register.

The size field contains the size of the operation.
If the size field is ØØ, it is a byte operation.
If the size field is Ø1, it is a word operation.
If the size field is 1Ø, it is a long word operation.

CMP
CoMPare address

CMP[l] .Ad, .As **Operand length(l): W, L**

```
CMP[l] .Ad, .As
       .Ad, .Ds
       .Ad, -@As
       .Ad, @As+
       .Ad, /exp[.W or .L]
       .Ad, [exp]@Ay[(Ri)]
       .Ad, exp[@PC[(Ri)]]
       .Ad, #exp
```

Example:

If AØ contains H'ØØØF FØØØ and Al contains H'ØØØF ØØØØ, then

```
CMPL     .AØ,.Al
```

clears the negative bit in the status register.

Instruction Fields:

15	14	13	12	11	1Ø	9	8	7	6	5	4	3	2	1	Ø
1	Ø	1	1	reg.			size			mode			reg.		

The register field contains the destination address register.

The size field contains the the size of the operation
If the size field is ØØ1, it is a word operation. The source is sign extended to a long operand and the operation is performed internally using all 32 bits. If the size field is Ø11, it is a long word operation.

The register and mode fields contain the source address mode.

CMP
CoMPare data

CMP Operand Length(1): B,W,L

```
CMP[1]     .Dd, .Ds
           .Dd, -@As
           .Dd, @As+
           .Dd, /exp [.W or .L]
           .Dd, [exp]@Ay[(Ri)]
           .Dd, exp[@PC[(Ri)]]
           .Dd, As  ([1]=W, L only)
```

Instruction Fields:

15	14	13	12	11	1Ø	9	8	7	6	5	4	3	2	1	Ø
1	Ø	1	1	reg.			size			mode			reg.		

The register field contains the destination data register.

The size field contains the size of the operation.
If the size field is ØØØ, it is a byte operation.
If the size field is ØØ1, it is a word operation.
If the size field is Ø1Ø, it is a long word operation.

The register and mode fields contain the source operand addressing mode.

DBcc
test condition Decrement and Branch

DBcc .Dx, exp[@PC] Operand length(1): W

A condition is tested. If the condition is determined to be true, no operation is performed. If the condition is false (not cc), the low order word (16 bits) of the data register is decremented. When false and the result is -1, then no other operation is performed (the program goes to the next instruction); if the result is anything besides -1, then the program counter is set to the value of the second operand (PC plus exp, where PC is the address of exp, the displacement word).

Examples:

If DØ contains H'Ø2, the zero bit in the status register is set, and LOOP is a statement label, then

```
DBNE     .DØ,LOOP
```

changes the contents of DØ to H'Ø1, and then transfers control of the program to the instruction at LOOP.

If DØ contains H'ØF1, the zero bit in the status register is clear, and LOOP is a statement label, then

```
DBNE     .DØ,LOOP
```

transfers control to the next sequential instruction (DØ is not decremented).

If DØ contains H'ØØ, the zero bit in the status register is set, and LOOP is a statement label, then

```
DBNE     .DØ,LOOP
```

transfers control to the next sequential instruction (DØ is decremented to -1).

Condition Codes:

X	N	Z	V	C
-	-	-	-	-

None of the flags are affected.

Instruction Fields:

This instruction requires one word of extension for displacement [exp].

15	14	13	12	11	1Ø	9	8	7	6	5	4	3	2	1	Ø
Ø	1	Ø	1	condition				1	1	Ø	Ø	1	reg.		
displacement															

The condition field contains one of 16 conditions.

The cc mnemonic can be one of the following:

Op Code	-	Mnemonic	-	Description
ØØØØ	-	DBR	-	always
Ø111	-	DBE	-	equal
Ø11Ø	-	DBNE	-	not equal
Ø1Ø1	-	DBC	-	carry
Ø1ØØ	-	DBNC	-	no carry
1Ø1Ø	-	DBP	-	positive
1Ø11	-	DBN	-	negative
1ØØ1	-	DBV	-	overflow
1ØØØ	-	DBNV	-	no overflow
111Ø	-	DBGT	-	greater than
11ØØ	-	DBGE	-	greater than or equal
11Ø1	-	DBLT	-	less than
1111	-	DBLE	-	less than or equal
ØØ1Ø	-	DBH	-	higher than
ØØ11	-	DBNH	-	not higher than
ØØØ1	-	DEC	-	never
Ø1Ø1	-	DBLO	-	low
Ø1ØØ	-	DBHS	-	high, or same

The register field contains the data register which is used as a counter.

The displacement field contains the 16 bit displacement (exp) and specifies the distance of the branch.

DIV
DIVide signed

```
DIV .Dd,  -@As
    .Dd,  @As+
    .Dd,  /exp [.W or .L]
    .Dd,  [exp]@Ay[(Ri)]
    .Dd,  exp[@PC[(Ri)]]
    .Dd,  #exp[.W or .L]
    .Dd,  .Ds
```

Operand length(1): W

Divides the destination (always a data register) by the source and the result is stored in the destination. The destination is a long word (32 bits) and the source is a word (16 bits). The division is performed using signed arithmetic. The result is a long word (32 bits) where:

1. The quotient is in the lower word.
2. The remainder is in the upper word.
3. The sign of the remainder is the same as the dividend unless the remainder is zero.

Special Conditions:

Division by zero causes a trap.

Overflow may be detected and set before completion of the operation.

If overflow occurs, the flag is set but the operands are unaffected.

Example:

If DØ contains H'ØØ14 and D1 contains -6 (H'FFFA), then

```
DIV     .DØ,.D1
```

changes the contents of DØ to H'ØØØ2 FFFD (-3 with a remainder of 2).

Condition Codes:

X	N	Z	V	C
-	*	*	*	*

X - Not affected.
N - Set if the quotient is negative, cleared otherwise, undefined if an overflow.
Z - Set if the quotient is zero, cleared otherwise, undefined if overflow.
V - Set if overflow detected, cleared otherwise.
C - Always cleared.

Instruction Fields:

15	14	13	12	11	1Ø	9	8	7	6	5	4	3	2	1	Ø
1	Ø	Ø	Ø		reg.		1	1	1		mode			reg.	

The register field contains the data (destination) register.

The register and mode fields contain the source address mode.

DIVU
DIVide Unsigned

```
DIVU .Dd, -@As                    Operand length(l): W
     .Dd, @As+
     .Dd, /exp [.W or .L]
     .Dd, [exp]@Ay[(Ri)]
     .Dd, exp[@PC[(Ri)]]
     .Dd, #exp[.W or .L]
     .Dd, .Ds
```

Divides the destination (always a data register) by the source and stores the result in the destination. The destination is a long word (32 bits) and the source is a word (16 bits). The division is performed using unsigned arithmetic. The result is a long word (32 bits) where:

1. The quotient is in the lower word.

2. The remainder is in the upper word.

Special Conditions:

Division by zero causes a trap.

Overflow may be detected and set before completion of the operation.

If overflow occurs, the flag is set, but the operands are unaffected.

Example:

If DØ contains H'ØØ14 and Dl contains H'ØØØ4, then

```
DIVU     .DØ,.Dl
```

changes the content of DØ to H'ØØØØ ØØØ5.

Condition Codes:

X	N	Z	V	C
–	*	*	*	*

X - Not affected.
N - Set if the most significant bit of the quotient is set, cleared otherwise, undefined if an overflow.
Z - Set if the quotient is zero, cleared otherwise, undefined if an overflow.
V - Set if an overflow is detected, cleared otherwise.
C - Always cleared.

Instruction Fields:

15	14	13	12	11	1Ø	9	8	7	6	5	4	3	2	1	Ø
1	Ø	Ø	Ø	reg.			Ø	1	1	mode			reg.		

The register fields contain the data (destination) register.

The register and mode fields contain the source address mode.

EXT
sign EXTended

EXT[l] .Dn **Operand length(l): W, L**

Sign-extends a byte to a word (bit 7 copied in bits 15-8), or a word to long word (bit 15 copied in bits 31-16).

The operand is always a data register.

Example:

If DØ contains H'ØØØØ FØØF, then

```
     EXTL     .DØ
```

changes the contents of DØ to H'FFFF FØØF.

Condition Code:

X	N	Z	V	C
-	*	*	Ø	Ø

X - Not affected.
N - Set if the result is negative, cleared otherwise.
Z - Set if the result is zero, cleared otherwise.
V - Always cleared.
C - Always cleared.

Instruction Fields:

15	14	13	12	11	1Ø	9	8	7	6	5	4	3	2	1	Ø
Ø	1	Ø	Ø	1	Ø	Ø	size			Ø	Ø	Ø	reg.		

The size field contains the size of the sign-extension.
If the size field is Ø1Ø, the sign extension is low order byte to word.
If the size field is Ø11, the sign extension is low order word to long word.

The register field contains the data register number (Ø-7) to be sign-extended.

LD
LoaD data

Can be interpretted by the Assembler-16 as four different instructions. By the operands used, the Assembler-16 chooses which instruction to initiate.

LD [1] **destination, source**

where the destination is a register and the second operand is the data located at source.

LD
LoaD condition codes

```
LD[l]    .CCR, #exp[.W]            Operand length(l): W
         .CCR, -@As
         .CCR, @As+
         .CCR, /exp[.W]
         .CCR, [exp]@As[(Ri)]
         .CCR, exp[@PC[(Ri)]]
         .CCR, .Ds
```

Loads the content of the source in the condition codes. The source is a word but only the low order 8 bits are loaded.

Example:

If the condition codes of the status register are all set, then

```
     LDW     .CCR,#H'ØE
```

changes the codes so that the extend and carry bits are clear, and the negative, zero, and overflow bits are set.

Condition Codes:

X	N	Z	V	C
*	*	*	*	*

Set all flags according to the source operand.

Instruction Fields:

15	14	13	12	11	1Ø	9	8	7	6	5	4	3	2	1	Ø
Ø	1	Ø	Ø	Ø	1	Ø	Ø	1	1	mode			reg.		

The <u>register</u> and <u>mode</u> <u>fields</u> contain the addressing modes of the source.

LD
LoaD data register

```
LD[l]   .Dd, -@As                  Operand length(l): B, W, L
        .Dd,  @As+
        .Dd, /exp[.W or .L]
        .Dd, [exp]@As[(Ri)]
        .Dd, exp[@PC[(Ri)]]
        .Dd, #exp[.W or .L]
        .Dd, .Ds
        .Dd, .As    (l=W, L only)
```

Loads the contents of the source into a destination data register.

Example:

If AØ points to memory address H'5ØØØ, which contains H'1F, and DØ contains H'ØØØØ ØØØØ, then

```
LDB      .DØ,@AØ+
```

changes the contents of DØ to H'ØØØØ ØØ1F, and increments AØ by 1.

Condition Codes:

X	N	Z	V	C
-	*	*	Ø	Ø

X = Unaffected.
N = Set if the result is negative, cleared otherwise.
Z = Set if the result is zero, cleared otherwise.
V = Always cleared.
C = Always cleared.

Instruction Fields:

15	14	13	12	11	1Ø	9	8	7	6	5	4	3	2	1	Ø
Ø	Ø	size		destination reg.			destination mode			source mode			source reg.		

The size field contains the size loaded.

If the size field contains Ø1 the size loaded is byte.
If the size field contains 11 the size loaded is word.
If the size field contains 1Ø the size loaded is long.

The destination fields determine the destination data register. Note that the register mode is reverse normal order.

The source fields determine the source addressing mode.

LD
LoaD address register

```
LD[l]  .Ad, #exp[W or .L]          Operand length(l): W, L
       .Ad, -@As
       .Ad, @As+
       .Ad, /exp[.W or .L]
       .Ad, [exp]@As[(Ri)]
       .Ad, exp[@PC[(Ri)]]
       .Ad, .As
       .Ad, .Ds
```

Loads the contents of the source to an address register.

Example:

If AØ contains H'ØØØØ ØØØØ, then

```
    LDW      .AØ,#H'FFØØ
```

changes the contents of AØ to H'FFFF FFØØ (the source is sign extended).

Condition Codes:

X	N	Z	V	C
-	-	-	-	-

None of the flags are affected.

Instruction Fields:

15	14	13	12	11	1Ø	9	8	7	6	5	4	3	2	1	Ø
Ø	Ø	size		destination reg.			Ø	Ø	1	source mode			source reg.		

The size field contains the size of the operand.
If the size field contains 11, it is a word operation. The source is sign extended to a long operand and all 32 bits are loaded into the address register.
If the size field contains 1Ø, it is a long operation.

The destination field contains the destination address register.

The source field contains the addressing mode of the source.

LDA
LoaD Address

```
LDA     .Ad, /exp[.W or .L]          Operand length(1): L
        .Ad, [exp]@As[(Ri)]
        .Ad, exp[@PC[(Ri)]]
```

Loads the specified address register with the address of the source. All 32 bits of the address register are affected.

Example:

If AØ contains H'ØØØØ ØØØØ, A1 contains H'6ØØØ and A2 contains H'ØØØØ ØØ25, then

```
     LDA       .AØ,H'1Ø@A1(A2)
```

Changes the contents of AØ to H'ØØØØ6ØØ35.

Condition Codes:

X	N	Z	V	C
-	-	-	-	-

None of the flags are affected.

Instruction Fields:

15	14	13	12	11	1Ø	9	8	7	6	5	4	3	2	1	Ø
Ø	1	Ø	Ø	reg.			1	1	1	mode			reg.		

The register field determine the address register to load.

The register and mode fields contain the address to be loaded.

LDM
LoaD Multiple

```
LDM[l] Rlist, @As+                Operand length(l): W, L
       Rlist, /exp
       Rlist, [exp]As([Ri)]
       Rlist, exp[@PC[(Ri)]]
```

where R list is a set of registers (destination), separated by commas (Rx, Ry...etc).

The registers in Rlist are loaded from consecutive memory locations beginning with the location specified by the source operand. The order of loading register is from DØ to D7, then from AØ to A7. Note that this order is independent of the order given in Rlist (.Al,.D3, D2 would give the same result as .D2, .D3, .Al.). If a word is stipulated in operand length (l), then the low order word of each register is loaded, and the word is sign extended into the upper word.

If the source is the postincrement mode, the incremented address register is updated to contain the address of the last word loaded plus two.

Example:

If the memory addresses H'6ØØØ-6ØØ3 contain H'AA AA BB BB, and DØ and Dl both contain H'ØØØØ ØØØØ, then

```
LDMW     .DØ,.Dl,/H'6ØØØ
```

changes the contents of DØ to H'ØØØØ AAAA, and Dl to H'ØØØØ BBBB.

Condition Codes:

X	N	Z	V	C
-	-	-	-	-

None of the flags are affected.

Instruction Fields:

A word extension is added to the operation word for this instruction (Rlist).

15	14	13	12	11	1Ø	9	8	7	6	5	4	3	2	1	Ø
Ø	1	Ø	Ø	1	1	Ø	Ø	1	sz	mode			reg.		
Rlist															

The size field contains the size of the operation.
If the size field contains Ø, it is word.
If the size field contains 1, it is long.

The register and mode fields contain the source addressing mode.Ø

The Rlist field contains the registers in the Rlist as follows:

15	14	13	12	11	1Ø	9	8	7	6	5	4	3	2	1	Ø
A7	A6	A5	A4	A3	A2	A1	AØ	D7	D6	D5	D4	D3	D2	D1	DØ

This is where the bits corresponding to the registers included in the Rlist are set.

LDP
LoaD Peripheral data

LDP[l] .Dd, [exp]@As **Operand length(l): W, L**

Loads the data into the data register (destination) from memory (source). The data in memory is formatted as one byte per word (the high order byte for even addresses and the low order byte for odd addresses).

Example:

If AØ contains H'6ØØØ, addresses H'6ØØØ-6ØØ3 contain H'FA 23 1B 3Ø, and DØ contains H'ØØØØ ØØØØ, then

```
LDPW      .DØ,@AØ
```

changes the contents of DØ to H'ØØØØ FA1B.

OPERAND LENGTH: W

MEMORY / SOURCE ADDRESS TO DATA REGISTER

A	B
C	D
E	F
G	H

EVEN --> | | | A | C |

ODD --> | | | B | D |

Example:

If AØ contains H'6ØØØ, addresses H'6ØØØ - 6ØØ7 contain H'FA 23 1B 3Ø 25 26 27 28, and DØ contains H'ØØØØ ØØØØ, then

```
LDPL      .DØ,@aØ
```

changes DØ to H'FA1B 25 27

OPERAND LENGTH: L

MEMORY / SOURCE ADDRESS TO DATA REGISTER

A	B
C	D
E	F
G	H

EVEN --> | A | C | E | G |

ODD --> | B | D | F | H |

Condition Codes:

X	N	Z	V	C
-	-	-	-	-

None of the flags are affected.

Instruction Fields:

15	14	13	12	11	1Ø	9	8	7	6	5	4	3	2	1	Ø
Ø	Ø	Ø	Ø	data reg.			1	Ø	sz	Ø	Ø	1	add reg.		
[exp]															

The data register field contains the destination of the data register.

The size field contains the size of the operation.
If the size field contains Ø, the operation is word.
If the size field contains 1, the operation is long.

The address register field contains the source address register used in indirect mode (plus optional displacement).

The [exp] field contains the placement used in calculating the operand address.

LINK
LINK and allocate

LINK .An, #exp **Operand length(l): Unsized**

This is a three-step instruction:

1. The address register specified (.An) is pushed onto the stack (.An occupied two words -- 32 bits).

2. Then, the address register (.An) is loaded with the updated stack pointer.

3. The sign-extended displacement (two's complement integer) is added to the stack pointer (#exp, occupies a 16-bit extension of the operation word).

Example:

If AØ contains H'ØØ11 4ØØØ, and the SP contains H'ØØØØ 3ØØ6, then

```
LINK          .AØ, #H'1ØØ
```

changes the contents of AØ to H'ØØØØ 3ØØ2, the SP to H'ØØØØ 3ØØ2, and memory addresses H'3ØØ2-3ØØ5 to H'ØØ 11 4Ø ØØ.

Condition Codes:

X	N	Z	V	C
-	-	-	-	-

None of the flags are affected.

Instruction Fields:

15	14	13	12	11	1Ø	9	8	7	6	5	4	3	2	1	Ø
Ø	1	Ø	Ø	1	1	1	Ø	Ø	1	Ø	1	Ø	reg.		
#exp															

The register field contains the address register specified in the operand.

The #exp field determines the two's complement integer which is to be added to the stack pointer.

MOV
MOVe

This instruction can be interpreted by the instruction four different ways. The operands used determine which instruction the Assembler-16 chooses.

General Operation:

```
MOV[l]    destination, source
```

where the operands are either both memory or both registers. The contents of the source is moved to the destination.

MOV
MOVe address register

MOV[l] .Ad, .As
       .Ad, .Ds

Operand length(l): W, L

The destination is an address register.

Example:

If A1 contains H'FFØØ, and AØ contains H'ØØØØ ØØØØ, then

    MOVW.AØ, .A1

changes the contents of AØ to H'FFFF FFØØ (the source is sign extended).

Condition Codes:

X	N	Z	V	C
-	-	-	-	-

None of the flags are affected.

Instruction Fields:

15	14	13	12	11	1Ø	9	8	7	6	5	4	3	2	1	Ø
Ø	Ø	size		destination register			Ø	Ø	1	source mode			source reg.		

The size field contains the size of the operand.
If the size field contains 11, it is a word operation. The source is sign extended to a long operand and all 32 bits are loaded into the address register.
If the size field contains 1Ø, it is a long operation.

The destination field determines the destination address register.

The source field determines the addressing mode of the source operand.

MOV
MOVe to condition codes

MOV[1] .CCR, .Ds **Operand length(1): W**

Loads the content of the source into the condition codes. Only the low order 8 bits of the source are loaded. The source is a word.

Example:

If DØ contains H'ØØ11, and the condition codes are all clear, then

```
MOV     .CCR, .DØ
```

sets the extend and carry bits of the status register, and clears the negative, overflow, and zero bits.

Condition Codes:

X	N	Z	V	C
*	*	*	*	*

All condition codes are set according to the source operand.

Instruction Fields:

15	14	13	12	11	1Ø	9	8	7	6	5	4	3	2	1	Ø
Ø	1	Ø	Ø	Ø	1	Ø	Ø	1	1	mode			reg.		

The register and mode fields contain the addressing mode of the source.

MOV
MOVe general

```
MOV[l]      .Dd, .Ds              Operand length(l): B, W, L
            .Dd, .As (l = W or L only)
           -@Ax, -@Ay
           @Ax+, @Ay+
           -@Ax, @Ay+
           -@Ax, /exp
           -@Ax, [exp]@Ay[(Ri)]
           -@Ax, exp[@PC[(Ri)]]
           @Ax+, -@Ay
           @Ax+, /exp
           @Ax+, [exp]@Ay[(Ri)]
           @Ax+, exp[@PC[(Ri)]]
           /exp, -@Ax
           /exp, @Ax+
          /expl, /exp2
          /expl, [exp2]@Ax[(Ri)]
          /expl, exp2[@PC[(Ri)]]
[exp]@Ax[(Ri)], -@Ay
[exp]@Ax[(Ri)], @Ay+
[expl]@Ax[(Ri)], /exp2
[expl]@Ax[(Ri)], [exp2]@Ay[(Ri)]
[expl]@Ax[(Ri)], exp2[@PC[(Ri)]]
           -@Ax, #exp
           @Ax+, #exp
          /expl, #exp2
[expl]@Ax[(Ri)], #exp2
```

Moves the contents of the source to the destination (memory to memory, or register to data register.

Example

If DØ contains H'1ØØØ and D1 contains H'FF, then

```
MOVB      .DØ,.D1
```

changes the contents of DØ to H'1ØFF

Condition Codes:

X	N	Z	V	C
-	*	*	Ø	Ø

X = Unaffected.
N = Set if result is negative, cleared otherwise.
Z = Set if result is zero, cleared otherwise.
V = Always cleared.
C = Always cleared.

Instruction Fields:

15	14	13	12	11	1Ø	9	8	7	6	5	4	3	2	1	Ø
				destination						source					
Ø	Ø	size		register			mode			mode			reg.		

The size field determines what size is moved.
If the size field is Ø1, it is byte.
If the size field is 11, it is word.
If the size field is 1Ø, it is long.

The destination fields contain the destination addressing mode (note reg./mode is reverse normal order).

The source fields contain the source addressing mode.

MOV
MOV from SR

MOV[l] .Dd, .SR Operand length(l): W

Moves the contents of the status register to the data register.

Example:

If the SR contains H'8715 and DØ contains H'ØØØØ, then

```
MOV         .DØ,.SR
```

changes the contents of DØ to H'8715.

Condition Codes:

X	N	Z	V	C
-	-	-	-	-

None of the flags are affected.

Instruction Fields:

15	14	13	12	11	1Ø	9	8	7	6	5	4	3	2	1	Ø
Ø	1	Ø	Ø	Ø	Ø	Ø	Ø	1	1	mode			reg.		

The register and mode fields contain the destination data register.

MUL
MULtiply signed

```
MUL  .Dd, -@As                    Operand length(l): W
     .Dd, @As+
     .Dd, /exp [.W or .L]
     .Dd, exp]@Ay[(Ri)]
     .Dd, exp[@PC[(Ri)]]
     .Dd, #exp[.W or .L]
     .Dd, .Ds
```

Multiplies the two signed word (16 bits) operands, producing a 32-bit signed result in the destination (data register). The register operands are taken from the low order word, leaving the high order word unused. All 32 bits of the product are saved in the destination.

Example:

If DØ contains H'ØØ1Ø and Dl contains H'FFF5, then

```
MUL       .DØ,.Dl
```

changes the contents of DØ to H'FFFF FF5Ø.

Condition Codes:

X	N	Z	V	C
-	*	*	Ø	Ø

X - Not affected.
N - Set if the result is negative, cleared otherwise.
Z - Set if the result is zero, cleared otherwise.
V - Always cleared.
C - Always cleared.

Instruction Fields:

15	14	13	12	11 10 9	8	7	6	5 4 3	2 1 Ø
1	Ø	Ø	Ø	register	1	1	1	mode	reg.

The register field contains the data register (destination).

The register and mode fields contain the source address mode.

MULU
MULtiply Unsigned

```
MULU .Dd, -@As                      Operand length(l): W
     .Dd, @As+
     .Dd, /exp [.W or .L]
     .Dd, [exp]@Ay[(Ri)]
     .Dd, exp[@PC[(Ri)]]
     .Dd, #exp[.W or .L]
     .Dd, .Ds
```

Multiplies two unsigned word (16 bits) operands, producing a 32-bit unsigned result in the destination (data register). The register operands are taken from the low order word; the high order word is unused. All 32 bits of the product are saved in the destination.

Example:

If DØ contains H'ØØ1Ø and D1 contains H'ØØØ5, then

```
MULU    .DØ, .D1
```

changes the contents of DØ to H'ØØØØ ØØ5Ø.

Condition Codes:

X	N	Z	V	C
-	*	*	Ø	Ø

X - Not affected.
N - Set if the most significant bit of the result is set, cleared otherwise.
Z - Set if the result is zero, cleared otherwise.
V - Always cleared.
C - Always cleared.

Instruction Fields:

15	14	13	12	11	1Ø	9	8	7	6	5	4	3	2	1	Ø
1	1	Ø	Ø	register			Ø	1	1	mode			reg.		

The register field contains the data register (destination).

The register and mode fields contain the source address mode.

NEG
NEGate

```
NEG[l]  .Dd                          Operand length(l): B, W, L
        -@Ad
        @Ad+
        /exp[.W or .L]
        [exp]@Ax[(Ri)]
```

Subtracts the operand addressed as the destination from zero. The result is stored in the destination.

Example:

If DØ contains H'3A, then

```
NEGB     .DØ
```

changes the contents of DØ to H'C6.

Condition Codes:

X	N	Z	V	C
*	*	*	*	*

X - Set the same as carry ((c) - if borrow generated), cleared otherwise.
N - Set if the result is negative, cleared otherwise.
Z - Set if the result is zero, cleared otherwise.
V - Set if an overflow is generated, cleared otherwise.
C - Set if a borrow is generated, cleared otherwise.

Instruction Fields:

15	14	13	12	11	1Ø	9	8	7	6	5	4	3	2	1	Ø
Ø	1	Ø	Ø	Ø	1	Ø	Ø	size		mode			reg.		

The size field contains the size of the operation.
If the size field is Ø1, it is byte.
If the size field is Ø1, it is word.
If the size field is 1Ø, it is long.

The register and mode fields contain the destination address mode.

NEGC
NEGate with carry

```
NEGC[l]  .Dd                    Operand length(l): B, W, L

         -@Ad
         @Ad+
         /exp[.W or .L]
         [exp]@Ax[(Ri)]
```

Subtracts the operand addressed as the destination and the carry bit from zero. The result is stored in the destination.

Example:

If DØ contains H'34 and the extend bit of the status register is set, then

```
     NEGCB    .DØ
```

changes the contents of DØ to H'CB.

Condition Codes:

X	N	Z	V	C
*	*	*	*	*

X - Set the same as carry ((c) - if borrow generated), cleared otherwise.
N - Set if the result is negative, cleared otherwise.
Z - Cleared if the result is nonzero, unchanged otherwise.
V - Set if an overflow is generated, cleared otherwise.
C - Set if a borrow is generated, cleared otherwise.

Instruction Fields:

15	14	13	12	11	1Ø	9	8	7	6	5	4	3	2	1	Ø
Ø	1	Ø	Ø	Ø	Ø	Ø	Ø	size		mode			reg.		

The size field contains the size of the operation.
If the size field is ØØ, it is byte.
If the size field is Ø1, it is word.

If the size field is 1Ø it is long.

The register and mode fields contain the destination address mode.

NEGD
NEGate Decimal (BCD) with extend

```
NEGD -@Ad                     Operand sizes: B
     @Ad+
     /exp
     [exp]@Ax[(Ri)]
     .Dd
```

Subtracts the operand and the extend bit from zero. Binary coded decimal arithmetic is used, storing the result in the operand address. The ten's complement of the destination is produced if the extend bit is clear, if the extend bit is set the nine's complement of the destination is produced. This is a byte only operation.

Example:

If DØ contains 51 and the extend bit of the status register is clear, then

```
NEGD     .DØ
```

changes the contents of DØ to 49.

Condition Codes:

X	N	Z	V	C
*	U	*	U	*

X - Set if borrow (BCD) occurred, cleared otherwise (same as carry(c)).
N - Undefined.
Z - Cleared if the result is nonzero, unchanged otherwise.
V - Undefined
C - Set if borrow (BCD) occurred, cleared otherwise. (Same as extend (X)).

Instruction Fields:

15	14	13	12	11	1Ø	9	8	7	6	5	4	3	2	1	Ø
Ø	1	Ø	Ø	1	Ø	Ø	Ø	Ø	Ø	mode			reg.		

The register and mode fields contain the operand destination (the address of the operand.)

NOP
No OPeration

NOP **Operand length(1): Unsized**

No operation occurs. The program counter is incremented by two. Otherwise, the processor state is unaffected.

Example:

If the PC contains H'6ØØØ, then

```
NOP
```

changes the PC to H/6ØØ2.

Condition Codes:

X	N	Z	V	C
-	-	-	-	-

None of the flags are affected.

Instruction Fields:

15	14	13	12	11	1Ø	9	8	7	6	5	4	3	2	1	Ø
Ø	1	Ø	Ø	1	1	1	Ø	Ø	1	1	1	Ø	Ø	Ø	1

NOT
logical NOT

```
NOT(1)  .Dd                      Operand length(1): B, W, L
        -@Ax
        @Ax+
        /exp[.W or .L]
        [exp]@Ax[(Ri)]
```

The one's complement of the operand is taken and is stored in the destination.

Example

If DØ contains H'3C, then

```
     NOTB     .DØ
```

changes the contents of DØ to H'C3.

Condition Codes:

X	N	Z	V	C
-	*	*	Ø	Ø

X - Not affected.
N - Set if the result is negative, cleared otherwise.
Z - Set if the result is zero, cleared otherwise.
V - Always cleared.
C - Always cleared.

Instruction Fields:

15	14	13	12	11	1Ø	9	8	7	6	5	4	3	2	1	Ø
Ø	1	Ø	Ø	Ø	1	1	Ø	size		mode			reg.		

The size field contains the size of the oeration.
   If the size field is ØØ, it is byte.
   If the size field is Ø1, it is word.
   If the size field is 1Ø, it is long.

The register and mode fields contain the destination address mode.

OR
logical OR

This instruction can be interpreted by the Assembler-16 two differnt ways. By the operands used, the Assembler-16 chooses which instruction to execute.

General Operation:

```
OR[l]  destination, source
```

where the source is ORed to the destination and the result is stored in the destination.

The Assembler-16 chooses which instruction to initiate by the following guidelines.

OR immediate	used if the source is immediate (indicated by a # sign).
OR data	used for all other OR operations, one of the operands is always a data register.

Condition Codes: (Identical for both operations)

X	N	Z	V	C
-	*	*	Ø	Ø

X - Not affected.
N - Set if the most significant bit of the result is set, cleared otherwise.
Z - Set if the result is zero, cleared otherwise.
V - Always cleared.
C - Always cleared.

OR
logical OR immediate

```
OR[l]         .Dd, #exp[.W or .L]  Operand length(l): B, W, L
             -@Ax, #exp[.W or .L]
             @Ax+, #exp[.W or .L]
/expl[.W or .L], #exp2[.W or .L]
[expl]@Ax[(Ri)], #exp2[.W or .L]
             .CCR, #exp[.W or .L] (l=W only)
```

Example:

If DØ contains H'C6, then

```
    ORB     .DØ, #H'2A
```

Changes the contents of DØ to H'EE.

Instruction Fields:

15	14	13	12	11	1Ø	9	8	7	6	5	4	3	2	1	Ø
Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	size		mode			reg.		

+

	byte data (8 bits)

or

word data (16 bits)

or

long word data (32 bits including previous word)

The size field contains the size of the operation.
If the size field is ØØ, it is byte.
If the size field is Ø1, it is word.
If the size field is 1Ø, it is long word.

The register and mode fields contain the address mode of the destination operand.

The data field contains the data immediately following the instruction:
If the size field is ØØ, then the data is the low order byte of the immediate word (8 bits).
If the size field is Ø1, then the data is the entire

immediate word (16 bits).
If the size field is 1Ø, then data is the next two immediate words (32 bits).

OR
Logical OR data

Operand length(l):B, W, L

```
OR[l]   .Dd, .Ds
        .Dd, -@Ay
        -@Ax , .Ds
        .Dd, @Ay+
        @Ax+, .Ds
        .Dd, /exp[.W or .L]
        /exp[.W or .L], .Ds
        .Dd, [exp]@Ay[(Ri)
        [exp]@Ax[(Ri)], .Ds
        .Dd, exp[@PC[(Ri)]]
```

Example:

If DØ contains H'C6 and Dl contains H'2A, then

```
ORB     .DØ, .Dl
```

changes the contents of DØ to H'EE.

Instruction Fields:

15	14	13	12	11	1Ø	9	8	7	6	5	4	3	2	1	Ø
1	Ø	Ø	Ø	reg.			size/op			mode			reg.		

The register field contains the data register.

The size/op field contains the size of the operation and the destination of the result:

Byte	Word	Long Word	Destination
ØØØ	ØØ1	Ø1Ø	data register
1ØØ	1Ø1	11Ø	second operand

The mode and register fields contain the location of the second operand.

PUSHA
PUSH Address

```
PUSHA   /exp                          Operand length(l): L
       [exp] @Ax [(Ri)]
        exp [@PC [(Ri)]]
```

Computes the address and pushes it into the stack.

Example:

If the SP contains H'ØØØØ 6ØØ4, then

```
     PUSHA      H'ØØFFØAØØ
```

changes the contents of memory locations H'6ØØØ-6ØØ3 to H'ØØ FF ØA ØØ, and the SP to H'ØØØØ 6ØØØ.

Condition Codes:

X	N	Z	V	C
-	-	-	-	-

None of the flags are affected.

Instruction Fields:

15	14	13	12	11	1Ø	9	8	7	6	5	4	3	2	1	Ø
Ø	1	Ø	Ø	1	Ø	Ø	Ø	Ø	1	mode			reg.		

The register and mode fields contain the address to be pushed onto the stack.

RET
RETurn from subroutine

RET **Operand length(1): Unsized**

Pops a long word from the stack and stores it in the program counter (PC). The previous program counter is lost.

Example:

If the top of the stack contains H'ØØ ØØ 6Ø ØØ, then

```
RET
```

changes the contents of the PC to H'6ØØØ and resumes execution from that address.

Condition Codes:

X	N	Z	V	C
-	-	-	-	-

None of the flags are affected.

Instruction Fields:

15	14	13	12	11	1Ø	9	8	7	6	5	4	3	2	1	Ø
Ø	1	Ø	Ø	1	1	1	Ø	Ø	1	1	1	Ø	1	Ø	1

RTR
ReTurn with Restore

RTR Operand length(1): Unsized

Pops a long word and word off the stack and then stores them in the program counter(PC) and condition code register (CCR), respectively. The previous values of the PC and CCR are lost. The Supervisor portion of the SR is unaffected.

Example:

If the top of the stack contains H'ØØ ØØ 6Ø ØØ ØØ Ø5, then

```
     RTR
```

sets all of the condition codes of the status register and changes the contents of the PC to H'6ØØØ where program execution resumes.

Condition Codes:

X	N	Z	V	C
*	*	*	*	*

All flags are set according to the word on the stack.

Instruction Fields:

15	14	13	12	11	1Ø	9	8	7	6	5	4	3	2	1	Ø
Ø	1	Ø	Ø	1	1	1	Ø	Ø	1	1	1	Ø	1	1	1

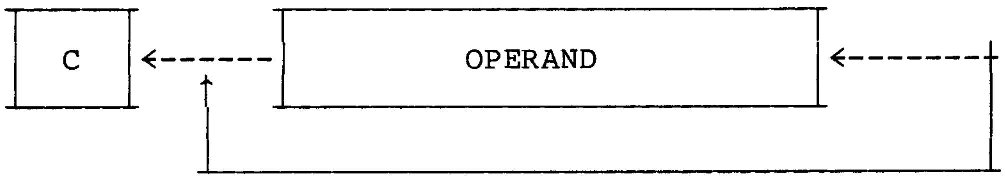
ROdc
ROtate

General Operation:

Rotates the bits of a specified data register by a count contained in either a second data register or an immediate expression whose value is in the range 1-8. Memory addresses of word length can also be rotated, but only by one bit. The direction and category are specified in the mnemonics:

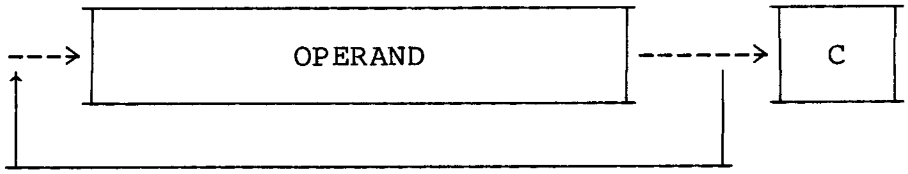
ROL - left logical
ROR - right logical
ROLC - left with carry(extend)
RORC - right with carry(extend)

ROL
ROtate Left logical



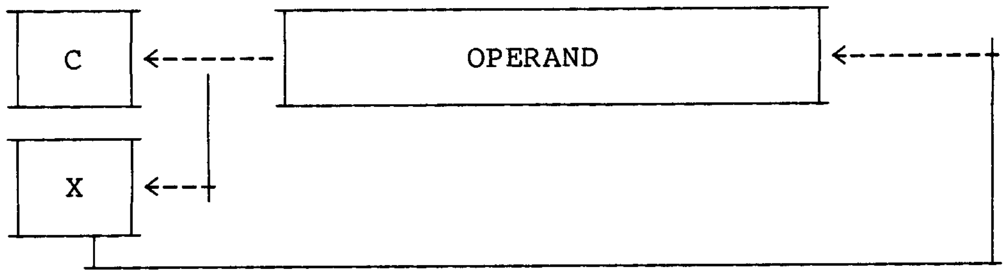
Bits rotated out of the high order bit go to the carry and low order bits. The extend bit is not modified.

ROR
ROtate Right logical



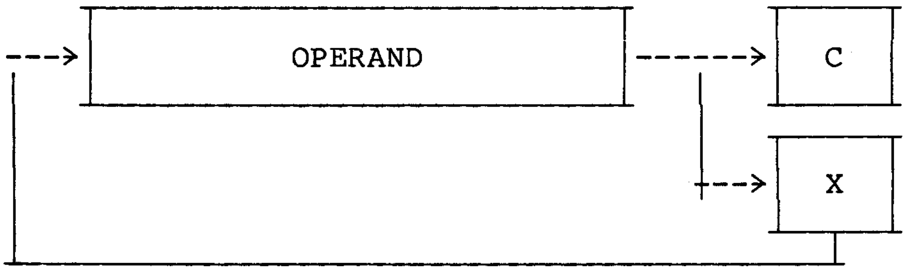
Bits rotated out of the low order bit go to the carry and high order bits. The extend bit is not modified.

ROLC
Rotate Left with Carry (extend)



Bits rotated out of the high order bit go to the carry and extend bits. The previous value of extend bit is rotated low order bit.

RORC
Rotate Right with Carry (extend)



Bits rotated out of the low order bit go to the carry and extend bits. The previous extend bit is rotated to low order bit.

ROL or ROR
ROtate logical

Condition Codes:

X	N	Z	V	C
-	*	*	Ø	*

X - Not affected.
N - Set if the most significant bit of the result is set, cleared otherwise.
Z - Set if the result is zero, cleared otherwise.
V - Always cleared.
C - Set according to the last bit rotated out of the operand, cleared for a shift count of Ø.

ROL or ROR
ROtate logical data

```
ROL[l] or                     Operand length(l):B, W, L
ROR[l]  .Dx, .Dy
        .Dx, #exp[.W or .L]
```

Example:

If DØ contains H'8ØØØ ØØØØ, then

```
     ROLL     .DØ, #1
```

changes the contents of DØ to H'ØØØØ ØØØ1.

Instruction Fields

15	14	13	12	11	1Ø	9	8	7	6	5	4	3	2	1	Ø
1	1	1	Ø	count			dr	size		i/r	1	1	reg.		

The count field contains the expression specifying the count.

If the immediate/register field contains Ø, the rotate count is specified in this field.
Values are 1-8 where 8 is indicated by ØØØ; values 1-7 sre standard binary, where ØØ1 = 1, Ø1Ø = 2 etc.

The direction field contains the direction of the rotation.

If the direction field contains Ø, the rotation is to the right.
If the direction field contains 1, the rotation is to the left.

The size field contains the size of the operation.

If the size field is ØØ, it is byte.
If the size field is Ø1, it is word.
If the size field is 1Ø, it is long word.

The immediate/register field contains the immediate/register.

If the immediate register field contains Ø, the count field contains an expression.
If the immediate register field contains 1, the count

field contains a register.

The register field contains the specifying register to be rotated.

ROL or ROR
ROtate logical memory

ROL[l] **Operand length(l): W**

```
ROL[l]
       or
ROR[l] -@An
        @An+
        /exp
        [exp]@An[Ri]
```

NOTE: rotate of one bit only

Example:

If memory address H'5ØØØ contains H'ØØ42, then

```
     ROR      /H'5ØØØ
```

changes the contents of H'5ØØØ to H'ØØ21.

Instruction Fields:

15	14	13	12	11	1Ø	9	8	7	6	5	4	3	2	1	Ø
1	1	1	Ø	Ø	1	1	dr	1	1	mode			reg.		

The direction field contains the direction of rotation.
If the direction field contains Ø, the rotation is to the right.
If the direction field contains 1, the rotation is to the left.

The register and mode fields contain the operand to be rotated.

ROLC or RORC
ROtate with Carry

Condition Codes:

X	N	Z	V	C
*	*	*	Ø	*

X - Set according to the bit last rotated out of the operand, cleared otherwise.
N - Set if the most significant bit of the result is set, cleared otherwise.
Z - Set if the result is zero, cleared otherwise.
V - Always cleared.
C - Set according to the last bit rotated out of the operand, unaffected for a shift count of Ø.

ROLC[l] or RORC[l]
ROtate with Carry data

ROLC[l] or **Operand length(l):B,W,L**
RORC[l] .Dx, .Dy
.Dx, #exp[.W or .L]

Example:

If DØ contains H'Ø8 and the carry bit of the status register is set, then

```
RORCB     .DØ, #2
```

changes the contents of DØ to H'42.

Instruction Fields:

15	14	13	12	11	1Ø	9	8	7	6	5	4	3	2	1	Ø
1	1	1	Ø	count			dr	size		i/r	1	Ø	reg.		

The <u>count</u> <u>field</u> contains the expression signifying register.
If the immediate/register field contains Ø, the rotate count is specified in this field.
Values are 1-8 where 8 is indicated by ØØØ; values 1-7 are standard binary where ØØ1 = 1, Ø1Ø = 2 etc.

The <u>direction</u> <u>field</u> contains the direction of the rotation.
If the direction field contains Ø, the rotation is to the right.
If the direction field contains 1, the rotation is to the left.

The <u>size</u> <u>field</u> contains the size of the operation.
If the size field is ØØ, it is byte.
If the size field is Ø1, it is word.
If the size field is 1Ø, it is long word.

The <u>immediate/register</u> <u>field</u> contains the immediate/register.
If the immediate register field contains Ø, the count field contains an expression.

If the immediate register field contains 1, the count field contains a register.

The register field contains the register to be rotated.

ROLC or RORC
ROtate with Carry memory

```
ROLC[l]                         Operand length(l): W
     or
RORC[l] -@An
         @An+
         /exp
         [exp]@An[(Ri)]
```

NOTE: Rotation of one bit only

Example:

If the memory address H'6ØØØ contains H'FØØØ and the carry bit of the status register is set, then

```
RORC     /H'6ØØØ
```

changes the contents of address H'6ØØØ to H'FØØØ.

Instruction Fields:

15	14	13	12	11	1Ø	9	8	7	6	5	4	3	2	1	Ø
1	1	1	Ø	Ø	1	Ø	dr	1	1	mode			reg.		

The direction field contains the direction of the rotation.

If the direction field contains Ø, the rotation is to the right.

If the direction field contains 1, the rotation is to the left.

The register and mode fields contain the operand to be rotated.

SETcc
SET on condition

SETcc .Dd
-@Ad
@Ad+
/exp
[exp]@Ad[(Ri)]

Operand Length(1): Byte

Tests a condition and sets the destination (byte only) to all ones (H'FF) if the condition is true, or to zero if the condition is false.

Example:

If DØ contains H'ØF and the zero bit of the status register is set, then

```
SETE    .DØ
```

changes the contents of DØ to H'FF.

Condition Codes:

X	N	Z	V	C
-	-	-	-	-

None of the flags are affected.

Instruction Fields:

15	14	13	12	11	1Ø	9	8	7	6	5	4	3	2	1	Ø
Ø	1	Ø	1	condition				1	1	mode			reg.		

The condition field is one of 16 conditions.

The cc mnemonic can be one of the following:

Op code	- Mnemonic	- Description
ØØØØ	- SET	- always
Ø111	- SETE	- equal
Ø11Ø	- SETNE	- not equal

Ø1Ø1	- SETC	- carry
Ø1ØØ	- SETNC	- no carry
1Ø1Ø	- SETP	- positive
1Ø11	- SETN	- negative
1ØØ1	- SETV	- overflow
1ØØØ	- SETNV	- no overflow
111Ø	- SETGT	- greater than
11ØØ	- SETGE	- greater than or equal
11Ø1	- SETLT	- less than
1111	- SETLE	- less than or equal
ØØ1Ø	- SETH	- higher than
ØØ11	- SETNH	- not higher than
ØØØ1	- SETF	- never
Ø1Ø1	- SETLO	- low
Ø1ØØ	- SETHS	- high or same

The <u>register</u> and <u>mode</u> <u>fields</u> contain the destination byte where Ø or 255 is stored.

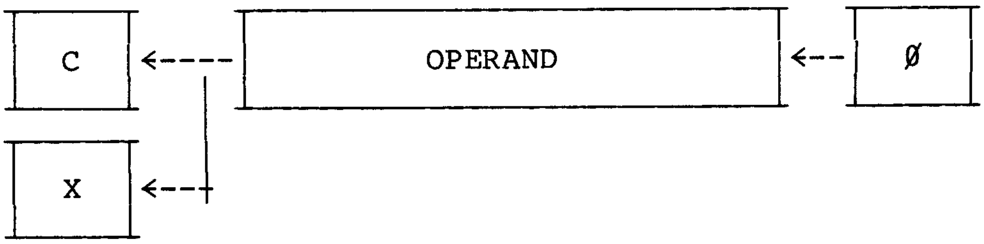
SHdc
SHift

General Operation:

Shifts the bits of a specified data register by an amount contained either in a second data register or an immediate expression whose value is in the range of 1-8. A memory address of word length can also be shifted, but only by one bit. In all cases the carry bit (C) as well as the extend bit (X) receive the bit shifted out of the operand. The direction and category are specified in the mnemonic as follows:

SHL - left logical
SHR - right logical
SHLA - left arithmetic
SHRA - right arithmetic

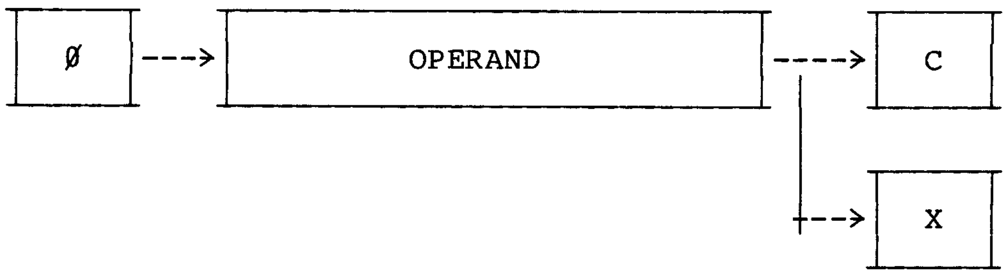
SHL and SHLA
SHift Left Logical/Arithmetic



Shifts the zeroes into the low order bit, and the high order bit into the carry and extend bit.

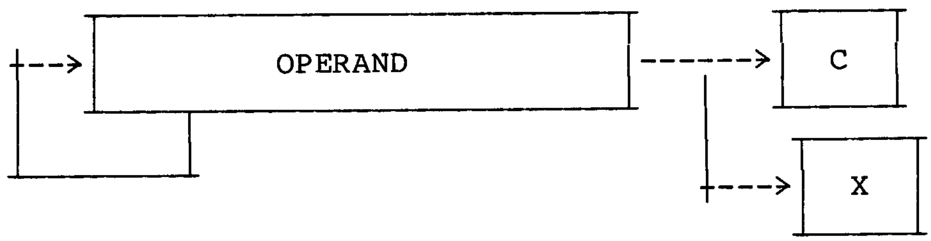
Note: The only difference in SHL and SHLA is in the overflow bit after the operation (see condition codes).

SHR
SHift Right Logical



Shifts the operand into the carry bit and the extension bit. Shifts a zero into the most significant bit.

SHRA
SHift Right Arithmetic



Replicates the sign bit and shifts the least significant bit into the carry and extend bits.

SHL or SHR
SHift logical

Condition Codes:

X	N	Z	V	C
*	*	*	Ø	*

X - Set according to the last bit shifted out of the operand, unaffected for a shift count of zero.
N - Set if the result is negative, cleared otherwise.
Z - Set if the result is zero, cleared otherwise.
V - Always cleared.
C - Set according to the last bit shifted out of the operand, cleared for a shift count of zero.

SHL or SHR
SHift logical data

SHL[l] Operand length(l):B,W,L
or
SHR[l].Dx, .Dy
.Dx, #exp[.W or .L]

Example:

If DØ contains H'8132, then

```
SHLW     .DØ, #1
```

changes the contents of DØ to H'Ø264.

If DØ contains H'8132, then

```
SHRW     .DØ, #1
```

changes the contents of DØ to H'4Ø99.

Instruction Fields:

15	14	13	12	11	1Ø	9	8	7	6	5	4	3	2	1	Ø
1	1	1	Ø	count			dr	size		i/r	Ø	1	reg.		

The count field contains the expression signifying the count.

If the immendiate/register field contains Ø, shift count is specified in this field. Values are 1-8 where 8 is indicated by ØØØ; values 1-7 are standard binary, where ØØ1 = 1 Ø1Ø = 2 etc.

The direction field contains the direction of the shift.

If the direction field contains Ø, the shift is to the right.
If the direction field contains 1, the shift is to the left.

The size field contains the size of the operation.

If the size field is ØØ, it is byte.
If the size field is Ø1, it is word.
If the size field is 1Ø, it is long word.

The immediate/register field contains the immediate/register.

If the immediate/register field contains Ø, the count field contains an expression.

If the immediate/register field contains 1, the count field contains a register.

The register field contains the register to be shifted.

SHift logical memory

SHL[l]
or
SHR[l]
-@An
@An+
/exp
[exp]@An[(Ri)]

Operand length(l): W

NOTE: Shift of one bit only

Example:

If memory address H'6ØØØ contains H'7341, then

```
     SHLW      /H'6ØØØ
```

changes the contents of address H'6ØØØ to H'E682.

If memory address H'6ØØØ contains H'7341, then

```
     SHRW      /H'6ØØØ
```

changes the contents of address H'6ØØØ to H'39AØ.

Instruction Fields:

15	14	13	12	11	1Ø	9	8	7	6	5	4	3	2	1	Ø
1	1	1	Ø	Ø	Ø	1	dr	1	1	mode			reg.		

The direction field contains the the direction of the shift.
If the direction field contains Ø, the shift is to the right.
If the direction field contains 1, the shift is to the left.

The register and mode fields contain the operand to be shifted.

SHLA or SHRA
SHift Arithmetic

Condition Codes:

X	N	Z	V	C
*	*	*	*	*

X - Set according to the last bit shifted out of operand, unaffected for a shift count of zero.
N - Set if the most significant bit of the result is set, cleared otherwise.
Z - Set if the result is zero, cleared otherwise.
V - Set if the most significant bit is changed at anytime during the operation; cleared otherwise.
C - Set according to the last bit shifted out of the operand; cleared for a shift count of zero.

SHLA[l]
SHift Arithmetic data

SHLAC[l] **or** **SHRA[l].Dx, .Dy** **.Dx, #exp[.W or .L]**

Operand length(l):B, W, L

Example:

If DØ contains H'8132, then

```
     SHLAW     .DØ,#1
```

changes the contents of DØ to H'Ø264.

If DØ contains H'8132, then

```
SHRAW     .DØ,#1
```

changes the contents of DØ to H'CØ99.

Instruction Fields:

15	14	13	12	11 10 9	8	7 6	5	4	3	2 1 Ø
1	1	1	Ø	count	dr	size	i/r	Ø	Ø	reg.

The count field contains the expression signifying the count.
If the immediate/register contains Ø, the shift count is specified in this field.
Values are 1-8 where 8 is indicated by ØØØ; values 1-7 are standard binary, where ØØ1 = 1, Ø1Ø = 2 etc.

The direction field contains the direction of the shift.
If the direction field contains Ø, the shift is to the right.
If the direction field contains 1, the shift is to the left.

The size field contains the size of the operation.
If the size field is ØØ, it is byte.
If the size field is Ø1, it is word.
If the size field is 1Ø, it is long word.

The immediate/register field contains the immediate/register.

If the immediate/register field contains Ø, the count field contains an expression.

If the immediate/register field contains 1, the count field contains a register.

The register field contains the register to be shifted.

SHLA or SHRA
SHift Arithmetic memory

```
SHLA[l]                          Operand length(l): W
     or
SHRA[l]  -@An
         @An+
         /exp
         [exp]@An[(Ri)]
```

NOTE: Shift of one bit only

Example:

If memory address H'6ØØØ contains H'7341, then

```
SHLAW     /H'6ØØØ
```

changes the contents of address H'6ØØØ to H'E682.

If memory address H'6ØØØ contains H'7341, then

```
SHRAW     /H'6ØØØ
```

changes the contents of address H'6ØØØ to H'39AØ.

Instruction Fields:

15	14	13	12	11	1Ø	9	8	7	6	5	4	3	2	1	Ø
1	1	1	Ø	Ø	Ø	Ø	dr	1	1	mode			reg.		

The direction field contains the direction of the shift.
If the direction field contains Ø, the shift is to the right.
If the direction field contains 1, the shift is to the left.

The register and mode fields contain the operand to be shifted.

ST
STore

This instruction can be interpreted by the Assembler-16 as two different instructions. By the operands used, the Assembler-16 chooses which instuction to initiate.

General Operation:

```
ST[1]    source, destination
```

where the source is the contents of a register and the destination is an address in memory or another register.

ST
STore data/address register

```
ST[l]    .Ds, -@Ad                   Operand length(l): B, W, L
         .Ds,  @Ad+
         .Ds, /exp[.W or .L]
         .Ds, [exp]@Ad[(Ri)]
         .Ds, .Ad
         .Ds, .Dd
         .As, -@Ad
         .As, @Ad+                 (Note: all operations using
         .As, /exp[.W or .L]       Address Register direct mode
         .As, [exp]@Ad[(Ri)]         have l = W or L only.)
         .As, .Ad
         .As, .Dd
```

Example:

If DØ contains H'FØØF and AØ contains H'6ØØØ, then

```
STW  .DØ,@AØ+
```

changes the contents of the memory address H'6ØØØ to H'FØØF and increments AØ by 2.

Condition Codes:

X	N	Z	V	C
-	*	*	Ø	Ø

X = Unaffected.
N = Set if the result is negative, cleared otherwise.
Z = Set if the result is zero, cleared otherwise.
V = Always cleared.
C = Always cleared.

Instruction Fields:

15	14	13	12	11	1Ø	9	8	7	6	5	4	3	2	1	Ø
Ø	Ø	size		destination reg.			destination mode			source mode			source reg.		

The size field contains the size loaded.

If the size field is Ø1, it is byte.
If the size field is 11, it is word.
If the size field is 1Ø, it is long.

The destination fields contain the destination addressing mode.

The source fields contain the source register.

ST
STore status register

```
ST[l]    .SR, Ad                      Operand length(l): W
         .SR, @Ad+
         .SR, /exp[.W]
         .SR, [exp]@Ad[(Ri)]
         .SR, .Dd
```

Example

If the SR contains H'8715, then

```
    ST        .SR,.DØ
```

changes the contents of DØ to H'8715.

Condition Codes:

X	N	Z	V	C
-	-	-	-	-

None of the flags are affected.

Instruction Fields:

15	14	13	12	11	1Ø	9	8	7	6	5	4	3	2	1	Ø
Ø	1	Ø	Ø	Ø	Ø	Ø	Ø	1	1	mode			reg.		

The register and mode fields contain the addressing mode of the destination.

STM
STore Multiple

```
STM[l]     Rlist, -@Ad              Operand length(l): W,L
           Rlist, /exp
           Rlist, [exp]@Ad[(Ri)]
```

Where Rlist is a set of registers (source) separated by commas (.Rx, .Ry...etc).

Stores the contents of the registers in Rlist in consecutive memory locations beginning with the location specified by the destination operand. The order of the register contents is DØ to D7 and AØ to A7; except in the predecrementing mode where the order is A7 to AØ, D7 to DØ. Note that this order holds independently of the order given in Rlist (.Al, .D3, .D2 gives the same results as .D2, .D3, and .Al). If a word is stipulated in the operand length(l), then the low order word of the register is stored.

If the destination operand is addressed in the predecrement mode, the registers are stored beginning with the specified address, minus two, and continues down through the lower addresses. The decremented address register is updated to contain the address of the last word stored.

Example:

If DØ contains H'FFFF and Dl contains H'ØØFF, then

```
STMW     .DØ, .Dl, /H'6ØØØ
```

changes the contents of memory addresses H'6ØØØ-6ØØ3 to H'FF FF ØØ FF.

Condition Codes:

X	N	Z	V	C
-	-	-	-	-

None of the flags are affected.

Instruction Fields:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	0	0	0	1	sz	mode			reg.		

The size field contains the size of the operation.
If the size field is 0, it is word.
If the size field is 1, it is long.

The register and mode fields contain the source addressing mode.

The Rlist field contains the registers in Rlist as follows:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
A7	A6	A5	A4	A3	A2	A1	A0	D7	D6	D5	D4	D3	D2	D1	D0

or, for the predecrement mode (where bits corresponding to the registers included in Rlist are set), the Rlist field contains the registers as follows:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
D0	D1	D2	D3	D4	D5	D6	D7	A0	A1	A2	A3	A4	A5	A6	A7

A word extension is added to the operation word for this instruction (Rlist).

STP
STore Peripheral

STP[l] .Ds, [exp]@Ad **Operand length(l): W, L**

Loads data from the data register (source) to alternate bytes of memory.

Examples:

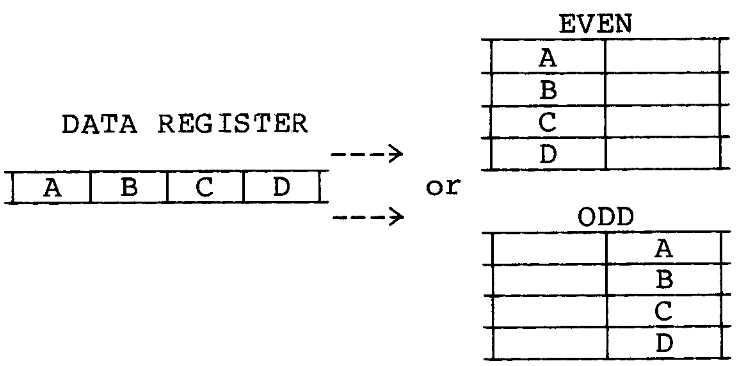
If DØ contains H'ØlFF 2FFl, and AØ contains H'6ØØØ, then

```
STPL      .DØ,@AØ
```

changes the contents of memory address H'6ØØØ to H'Øl, H'6ØØ2 to H'FF, H'6ØØ4 to H'2F, and H'6ØØ6 to H'Fl.

OPERAND LENGTH: L

DESTINATION ADDRESS

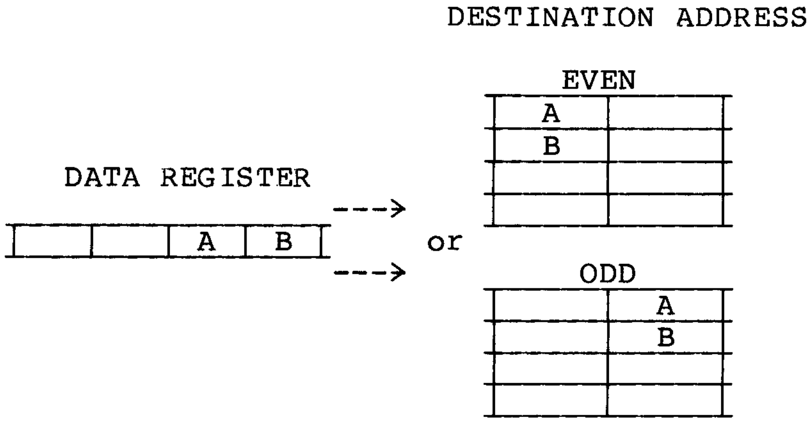


If DØ contains H'ØlFF and AØ contains H'6ØØØ, then

```
STPW      .DØ,@AØ
```

changes the contents of memory address H'6ØØØ to H'Øl and address H'6ØØ2 to H'FF.

OPERAND LENGTH: W



Condition Codes:

X	N	Z	V	C
-	-	-	-	-

None of the flags are affected.

Instruction Fields:

15	14	13	12	11	1Ø	9	8	7	6	5	4	3	2	1	Ø
Ø	Ø	Ø	Ø	data reg.			1	1	sz	Ø	Ø	1	add.reg.		
[exp]															

The size field contains the size of the operation.
    If the size field is Ø, it is word.
    If the size field is 1, it is long.

The data register field contains the source of the data register.

The address register field contains the address register used in the indirect mode (plus optional displacement).

The [exp] field specifies the displacement used in calculating the operand address.

SUB
SUBtract

This instruction can be interpreted by the Assembler-16 four different ways. The Assembler-16 specifies which is to be executed by the operands used.

General Operation:

```
SUB[l]  destination, source
```

where the source is subtracted from the destination (d) and the result is stored in the destination. The size of the operation will vary from one operation to another and is optionally stipulated.

The Assembler-16 chooses which instruction to execute according to the following guidelines:

SUB quick	source is immediate (indicated by a # sign) and in the value range 1-8.
SUB address	destination is an address register.
SUB immediate	source is immediate and greater than 8 (more than 3-bit-data).
SUB data register	all remaining subtraction operations. A data register is always one of the operands.

SUB
SUB quick/ SUB immediate

```
SUB[l]      .Dd , #exp[.W or .L]  Operand length(l): B, W, L
           -@Ad, #exp[.W or .L]
           @Ad+, #exp[.W or .L]
/expl[.W or .L], #exp2[.W or .L]
[expl]@Ax[(Ri)], #exp2[.W or .L]
            .Ad, #exp[.W or .L] (quick only)
```

Example:

If DØ contains H'3FØ5, then

```
SUBB    .DØ, #3
```

changes the contents of DØ to H'3FØ2.

Condition Codes:

X	N	Z	V	C
*	*	*	*	*

X - Set the same as carry, set if borrow is generated, cleared otherwise.
N - Set if the result is negative, cleared otherwise.
Z - Set if the result is zero, cleared otherwise.
V - Set if the overflow is generated, cleared otherwise.
C - Set if borrow is generated, cleared otherwise.

No flags are affected if Subtraction to Address register is made.

Instruction Fields: SUB quick

15	14	13	12	11	1Ø	9	8	7	6	5	4	3	2	1	Ø
Ø	1	Ø	1	data			1	size		mode			reg.		

The data field contains 3-bit data, values 1-8, (ØØ1-111 = 1-7 decimal; ØØØ=8 decimal)

The size field contains the size of the operation.
If the size field is ØØ, it is byte.

If the size field is Ø1, it is word.
If the size field is 1Ø, it is long word.

The register and mode fields contain the destination.
If the size field is byte then the address register direct is not allowed.
If the size field is word and the destination is the address Register, the source is sign-extended to 32-bits (see SUB address register).

Condition Codes:

X	N	Z	V	C
*	*	*	*	*

X - Set the same as carry(C), set if borrow is generated, cleared otherwise.
N - Set if the result is negative, cleared otherwise.
Z - Set if the result is zero, cleared otherwise.
V - Set if overflow is generated, cleared otherwise.
C - Set if borrow is generated, cleared otherwise.

Instruction Fields: SUB immediate
Operand length(1): B,W,L

The number of extensions for the immediate data vary according to the size of the operation (see data fields).

15	14	13	12	11	1Ø	9	8	7 6	5 4 3	2 1 Ø
Ø	Ø	Ø	Ø	Ø	1	Ø	Ø	size	mode	reg.

+

	byte data (8 bits)

or

word data (16 bits)

or

long word data (32 bits, including previous word)

The size field contains the size of the operation.
If the size field is ØØ, it is byte.
If the size field is Ø1, it is word.
If the size field is 1Ø, it is long word.

The register and mode fields contain the address mode of the destination operand.

The data fields contain the data immediately following the instruction:

If the size field is ØØ, the data is in the low order byte of the immediate word (8 bits).
If the size field is Ø1, the data is the entire immediate word (16 bits).
If the size field is 1Ø, the data is the next two immediate words (32 bits).

SUB
SUB address register

```
SUB[l]  .Ad, .As                           Operand length(l):W,L
        .Ad, .Ds
        .Ad, -@As
        .Ad, @As+
        .Ad, /exp[.W or .L]
        .Ad, [exp]@Ay[(Ri)]
        .Ad, exp[@PC[(Ri)]]
        .Ad, #exp[.W or .L](SUB immediate only)
```

Note that an address register is always the destination operand.

Example:

If AØ contains H'ØØ3F ØØ1Ø and A1 contains H'ØØØØ ØØØ4, then

```
SUBL    .AØ, .A1
```

changes the contents of AØ to H'ØØ3F ØØØC.

Condition Codes:

X	N	Z	V	C
-	-	-	-	-

None of the flags are affected.

Instruction Fields:

15	14	13	12	11	1Ø	9	8	7	6	5	4	3	2	1	Ø
1	Ø	Ø	1	reg.(d)			size			mode			reg.		

The register(d) field contains the destination operand. It can be any address register.

The size field contains the size of the operation.
If the size field is Ø11, it is word. The source operand is sign-extended (see EXT) to fill 32 bits of the address register.
If the size field is 111, it is long word.

The register and mode fields contain the address mode of the source operand.

SUB
SUB data register

Operand length(l):B, W, L

```
SUB[l]  .Dd, .Ds                   :        -@Ad ,.Ds
        .Dd, -@As                  :        @Ad+,.Ds
        .Dd, @As+                  :        /exp[.W or .L],.Ds
        .Dd, /exp[.W or .L]        :        [exp]@Ax[(Ri)],.Ds
        .Dd, [exp]@Ay[(Ri)]        :
        .Dd, exp[@PC[(Ri)]]        :
        .Dd, .As                   :
     operand list continued in next column
```

A data register is always one of the operands.

Example:

If DØ contains H'3F and Dl contains H'1Ø; then

```
SUBB     .DØ, .Dl
```

changes the contents of DØ to H'2F.

Condition Codes:

X	N	Z	V	C
*	*	*	*	*

X - Set if a borrow occurred, cleared otherwise (same as carry(C)).
N - Set if the result is negative, cleared otherwise.
Z - Set if the result is zero, cleared otherwise.
V - Set if an overflow is generated, cleared otherwise.
C - Set if a borrow occurred, cleared otherwise.

Instruction Fields:

15	14	13	12	11	1Ø	9	8	7	6	5	4	3	2	1	Ø
1	Ø	Ø	1	reg.			size/op			mode			reg.		

The <u>register field</u> contains the data register.
The <u>size/op field</u> contains the size of the operation and specifies the destination of the result:

Byte	Word	Long Word	Destination
ØØØ	ØØ1	Ø1Ø	data register
1ØØ	1Ø1	11Ø	second operand

The mode and register fields contain the location of the second operand.

If the second operand is the source operand and its size is one byte, the address register direct addressing mode is not permitted.

SUBC
SUBtract with Carry

SUBC[l] .Dd, .Ds **Operand Sizes: B, W, L**
-@Ad, -@As

Subtracts the source and the carry bit from the destination. The results are stored in the destination location.

Example:

If DØ contains H'3F, Dl contains H'1Ø, and the carry bit of the status register is set, then

```
SUBCB    .DØ, .Dl
```

changes the contents of DØ to H'2E.

Condition Codes:

X	N	Z	V	C
*	*	*	*	*

X - The same as carry; set if carry generated, cleared otherwise.
N - Set if the result is negative, cleared otherwise.
Z - Cleared if the result is nonzero, unchanged otherwise.
V - Set if overflow is generated, cleared otherwise.
C - Set if carry is generated, cleared otherwise.

Instruction Fields:

15	14	13	12	11	1Ø	9	8	7	6	5	4	3	2	1	Ø
1	Ø	Ø	1	reg.(d)			1	size		Ø	Ø	r/m	reg.(s)		

The R/M field contains the operand addressing mode.
If the R/M field is Ø, the operation is from data register to data register.
If the R/M field is 1, the operation is from memory to memory.

The size field contains the size of the operation.
If the size field is ØØ, it is byte.

If the size field is Ø1, it is word.
If the size field is 1Ø, it is long word.

SUBD
SUBtract Decimal (BCD) with extend

SUBD .Dd, .Ds
-@Ad, -@As

Operand sizes: B

Subtracts the source and the extend bit from the destination. The result is stored in the destination. Binary coded decimal arithmetic is used. The operation size is byte only.

Example:

If DØ contains 41, D1 contains 35, and the carry bit is clear, then

```
SUBD     .DØ, .D1
```

changes the contents of DØ to Ø6.

Condition Codes:

X	N	Z	V	C
*	U	*	U	*

X - Same as carry; set if borrow (BCD) occurs, cleared otherwise.
N - Undefined.
Z - Cleared if the result is nonzero, unchanged otherwise.
V - Undefined.
C - Set if borrow (BCD) occurs, cleared otherwise.

Instruction Fields:

15	14	13	12	11	1Ø	9	8	7	6	5	4	3	2	1	Ø
1	Ø	Ø	Ø	reg.(d)			1	Ø	Ø	Ø	Ø	r/m	reg.(s)		

The <u>register(d) field</u> contains the destination register.
If the R/M field is Ø, reg(d) is a data register.
If the R/M is 1, reg(d) is address register in the predecrement addressing mode.

The <u>R/M field</u> contains the operand addressing mode.

If the R/M field is Ø, the operation is from data register to data register.
If the R/M field is 1, the operation is from memory to memory.

The register(s) field contains the source register.
If the R/M field is Ø, register(s) is a data register.
If the R/M field is 1, register(s) is an address register in predecrement addressing mode.

TEST
TEST an operand

```
TEST(l) .Dn
        -@An
        @An+
        /exp[.W or .L]
        [exp]@An[(Ri)]
```

Operand length(l): B, W, L

Compares the operand to zero. Sets the condition codes according to the results. No results are saved.

Example:

If DØ contains H'ØØ, then

```
     TESTB     .DØ
```

sets the zero bit of the status register and clears all other status bits.

Condition Codes:

X	N	Z	V	C
-	*	*	Ø	Ø

X = Not affected.
N = Set if the operand is negative, cleared otherwise.
Z = Set if the operand is zero, cleared otherwise.
V = Always cleared.
C = Always cleared.

Instruction Fields:

15	14	13	12	11	1Ø	9	8	7	6	5	4	3	2	1	Ø
Ø	1	Ø	Ø	1	Ø	1	Ø	size		mode			reg.		

The size field contains the size of the operation.
If the size field is ØØ, it is byte.
If the size field is Ø1, it is word.
If the size field is 1Ø, it is long.

The register and mode fields contain the addressing mode of the operand being tested.

TESTSET
TEST and SET indivisible

```
TESTSET .Dn                          Operand length(l): B
        -@An
        @An+
        /exp[.W or .L]
        [exp]@An[(Ri)]
```

Tests the operand (byte) for negative and zero conditions, and then sets the most significant bit of the byte. The operation is indivisible (noninterruptable) in a multi-user environment.

Example:

If DØ contains H'ØØ, then

```
     TESTSET     .DØ
```

sets the zero bit of the status register and changes the contents of DØ to H'8Ø.

Condition Codes:

X	N	Z	V	C
-	*	*	Ø	Ø

X - Not affected.
N - Set if the most significant bit of the operand is set, cleared otherwise.
Z - Set if the operand zero, cleared otherwise.
V - Always cleared.
C - Always cleared.

Instruction Fields:

15	14	13	12	11	1Ø	9	8	7	6	5	4	3	2	1	Ø
Ø	1	Ø	Ø	1	Ø	1	Ø	1	1	mode			reg.		

The <u>register</u> and <u>mode</u> <u>fields</u> contain the operand addressing mode.

TEST1
TEST bit

General Operation:

Tests a bit (specified by the source) in the destination. The state of the tested bit is reflected in the Z condition code. If a data register is the destination, then the bit numbering is modulo 32(long word). If the destination is in memory, the operation is performed using modulo 8 (byte) and the contents of the byte are unchanged.

TEST1[1]   source(s) destination(d)

Condition Codes:

X	N	Z	V	C
-	-	*	-	-

X - Not affected.
N - Not affected.
Z - Set if the bit tested is zero, cleared otherwise.
V - Not affected.
C - Not affected.

TEST1 data register

```
TEST1[1] .Ds, -@Ay
         .Ds, @Ay+
         .Ds, /exp
         .Ds, [exp]@Ay[(Ri)]
         .Ds, exp[@PC[(Ri)]]
         .Ds, .Dd
```

Operand length(1): B for memory destination L for data register destination

The source is a data register (specified as the first operand).

Example:

If DØ contains H'ØØØØ ØØØ5 and D1 contains H'ØØØØ ØØØ4, then

```
TEST1L    .DØ, .DL
```

sets the zero bit of the status register.

Instruction Fields:

15	14	13	12	11	1Ø	9	8	7	6	5	4	3	2	1	Ø
Ø	Ø	Ø	Ø	reg.(s)			1	Ø	Ø	mode			reg.		

The register(s) field contains the data register which contains the bit number

The register and mode fields contain the destination operand.

TEST1 immediate

```
TEST1[1]    #exp, Dd                 Operand length(1): B for
            #exp, -@Ad               memory destination L for
            #exp, @Ad+               Data Register destination.
           #exp1, /exp2
           #exp1, [exp2]@Ad[(Ri)]
           #exp1, exp2[@PC[(Ri)]]
```

Note: source (first operand) is immediate, requires a second word of instruction:

Example:

If DØ contains H'Ø5, then

```
     TEST1L   #3, .DØ
```

sets the zero bit of the status register.

Instruction Fields:

15	14	13	12	11	1Ø	9	8	7	6	5	4	3	2	1	Ø
Ø	Ø	Ø	Ø	1	Ø	Ø	Ø	Ø	Ø	mode			reg.		
bit number															

The <u>register</u> and <u>mode</u> <u>fields</u> contain the destination.

The <u>bit</u> <u>number</u> <u>field</u> specifies the bit number.

TESTCLR1
TEST and CLeaR bit

General Operation:

Tests a bit (specified by the source) in the destination. The state of the tested bit is reflected in the Z condition code. After the test, the bit is cleared. If a data register is the destination, then the bit numbering is modulo 32 (long word). If the destination is in memory, the operation is performed using modulo 8 (byte) and the byte is written back to the location.

```
TESTCLR1[1]      source, destination
```

Condition Codes:

X	N	Z	V	C
-	-	*	-	-

X - Not affected.
N - Not affected.
Z - Set if the bit tested is zero, cleared otherwise.
V - Not affected.
C - Not affected.

TESTCLR1 data register

```
TESTCLR1[1] .Ds, -@Ay
            .Ds, @Ay+
            .Ds, /exp
            .Ds, [exp]@Ay[(Ri)]
            .Ds, .Dd
```

Operand length: B for memory destination. L for Data Register destination.

The source is a data register (specified as the first operand).

Example:

If DØ contians H'Ø5 and D1 contains H'1F, then

```
TESTCLR1L    .DØ, .D1
```

changes the contents of D1 to H'1F and sets the zero bit of the status register.

Instruction Fields:

15	14	13	12	11	1Ø	9	8	7	6	5	4	3	2	1	Ø
Ø	Ø	Ø	Ø	reg.(s)			1	1	Ø	mode			reg.		

The register(s) field contains the data register which contains the bit number.

The register and mode fields contain the destination operand.

TESTCLR1 immediate

```
TESTCLR1[1]  #exp, .Dd             Operand length(1): B for memory
             #exp, -@Ad                  destination. L for data
             #exp, @Ad+                  register destination.
            #exp1, /exp2
            #exp1, [exp2]@Ad[(Ri)]
```

The source (first operand) is immediate, and requires a second word of instruction:

Example:

If DØ contains H'FØ, then

```
    TESTCLR1L    #3, .DØ
```

sets the zero bit of the status register (DØ is unchanged).

Instruction Fields:

15	14	13	12	11	1Ø	9	8	7	6	5	4	3	2	1	Ø
Ø	Ø	Ø	Ø	1	Ø	Ø	Ø	1	Ø	mode			reg.		
bit number															

The register and mode fields contain the destination.

The bit number field specifies the bit number.

TESTNOT1
TEST and NOT bit

General Operation:

Tests a bit (specified by the source) in the destination. The state of the tested bit is reflected in the Z condition code. After the test, the bit is changed (NOT). If a data register is the destination, then the bit numbering is modulo 32 (long word). If the destination is in memory, the operation is performed using modulo 8 (byte) and the byte is written back to the location.

```
TESTNOT1[1]         source, destination
```

Condition Codes:

X	N	Z	V	C
-	-	*	-	-

X - Not affected.
N - Not affected.
Z - Set if the bit tested is zero, cleared otherwise.
V - Not affected.
C - Not affected.

TESTNOT1 data register

```
TESTNOT1[1] .Ds, -@Ay           Operand length(1): B for memory
            .Ds, @Ay+                destination. L for Data
            .Ds, /exp                Register destination
            .Ds, [exp]@Ay[(Ri)]
            .Ds, .Dd
```

The source is a data register (specified as the first operand).

Example:

If DØ contains H'Ø4 and D1 contains H'Ø4, then

```
TESTNOT1L    .DØ, .D1
```

changes the contents of D1 to H'14 and sets the zero bit in the status register.

Instruction Fields:

15	14	13	12	11	1Ø	9	8	7	6	5	4	3	2	1	Ø
Ø	Ø	Ø	Ø	reg.(s)			1	Ø	1	mode			reg.		

The register(s) field contains the data register which contains the bit number.

The register and mode fields contain the destination operand.

TESTNOT1 immediate

```
TESTNOT1[l]     #exp, .Dd        Operand length(l): B for memory
                #exp, -@Ad                 destination. L for Data
                #exp, @Ad+                 Register destination.
               #expl, /exp2
               #expl, [exp2]@Ad[(Ri)]
```

The source (first operand) is immediate, and requires a second word of instruction:

Example:

If DØ contains H'FØ, then

```
TESTNOT1L  #5, .DØ
```

changes the contents of DØ to H'DØ and sets the zero bit of the status register.

Instruction Fields:

15	14	13	12	11	1Ø	9	8	7	6	5	4	3	2	1	Ø
Ø	Ø	Ø	Ø	1	Ø	Ø	Ø	Ø	1	mode			reg.		
Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	bit number							

The register and mode fields contain the destination.

The bit number field contains the bit number.

UNLK
UNLinK

UNLK .An **Operand length(1): Unsized**

Undoes A LINK. Loads the stack pointer from the specified address register. The address register is loaded with the long word pulled from the top of the stack.

Example:

If AØ contains H'ØØØØFFØØ and the memory address H'ØØØØ FFØØ contains H'ØØ FA ØØ ØØ, then

```
UNLK     .AØ
```

changes the contents of AØ to H'ØØFA ØØØØ and the SP now contains H'ØØØØ FFØ4.

Condition Codes:

X	N	Z	V	C
-	-	-	-	-

None of the flags are affected.

Instruction Fields:

15	14	13	12	11	1Ø	9	8	7	6	5	4	3	2	1	Ø
Ø	1	Ø	Ø	1	1	1	Ø	Ø	1	Ø	1	1	reg.		

The <u>register</u> <u>field</u> contains the address register through which the unlinking is to be done.

XCH
eXCHange

This instruction can be interpreted by the Assembler-16 as two different instructions. The operands used determine which instruction is executed.

XCH[l].Ax, .Ay **eXCHange registers** **Operand length(l): L**
.Ax, .Dy
.Dx, .Dy
.Dx, .Ay

The contents of two Registers (32 bits) are exchanged.

Example:

If DØ contains H'FFØØ FFØØ and D1 contains H'12FF ØØØØ, then

```
    XCH     .DØ, .D1
```

changes the contents of DØ to H'12FF ØØØØ and D1 to H'FFØØ FFØØ.

Condition Codes:

X	N	Z	V	C
-	-	-	-	-

None of the flags are affected.

Instruction Fields:

15	14	13	12	11	1Ø	9	8	7	6	5	4	3	2	1	Ø
1	1	Ø	Ø	reg. X			1	op/mode					reg. Y		

The register X field contains one of the registers. If the exchange is between the data register and address register, the data register is specified here.

The op/mode field specifies the type of exchange.
 If the op/mode field is Ø1ØØØ, it is data registers.
 If the op/mode field is Ø1ØØ1, it is address registers.
 If the op/mode field is 1ØØØ1, it is both data register and address register.

The register Y field contains the second register. If the exchange is between address and data registers the address register is specified here.

XCH[1] .Dn
eXCHange words

Operand length(1): W

Exchanges the high order word (16 bits) and low order words (16 bits) in a data register.

Example:

If DØ contains H'FØØF ØØØØ, then

```
    XCH     .DØ
```

changes the contents of DØ to H'ØØØØ FØØF.

Condition Codes:

X	N	Z	V	C
-	*	*	Ø	Ø

X - Not affected.
N - Set if the most significant bit of the 32-bit result is set, cleared otherwise.
Z - Set if the 32-bit result is zero, cleared otherwise.
V - Always cleared.
C - Always cleared.

Instruction Fields:

15	14	13	12	11	1Ø	9	8	7	6	5	4	3	2	1	Ø
Ø	1	Ø	Ø	1	Ø	Ø	Ø	Ø	1	Ø	Ø	Ø	reg.		

The register field contains the data register used.

XOR
eXclusive OR logical

This instruction can be interpreted by the Assembler-16 as two different instructions. By the operands used, the Assembler-16 determines which instruction to execute.

General Operation:

XOR[l] destination(d), source(s)

where the source is eXclusive ORed to the destination. The result is stored in the destination.

The Assembler-16 chooses which instruction to initiate by the following guidelines:

XOR immediate	if the source is immediate (indicated by a # sign).
XOR data	if the source is a data register.

Condition Codes: (Identical for both operations)

X	N	Z	V	C
–	*	*	Ø	Ø

X - Not affected.
N - Set if the most significant bit of the result is set, cleared otherwise.
Z - Set if result is zero, cleared otherwise.
V - Always cleared.
C - Always cleared.

XOR
eXclusive OR data

```
XOR[1]        .Dd, .Ds
             -@Ax, .Ds
             @Ax+, .Ds
[exp]@Ax[(Ri)], .Ds
/exp[.W or .L], .Ds
```

Operand length(1): B, W, L

Example:

If DØ contains H'F5 and D1 contains H'Ø7, then

```
     XORB      .DØ, .D1
```

changes the contents of DØ to H'F2.

Instruction Fields:

15	14	13	12	11	1Ø	9	8	7	6	5	4	3	2	1	Ø
1	Ø	1	1	reg.(s)			size/op			mode			reg.		

The register(s) field contains the data register.

The size/op field is as below:

Byte	Word	Long Word
1ØØ	1Ø1	11Ø

The register and mode fields contain the destination operand.

XOR
eXclusive OR immediate

```
XOR[l]        .Dd, #exp[.W or .L]  Operand length(l): B, W, L
             -@Ax, #exp[.W or .L]
             @Ax+, #exp[.W or .L]
/expl[.W or .L], #exp2[.W or .L]
[expl]@Ax[(Ri)], #exp2[.W or .L]
             .CCR, #exp[.W or .L]  (l=B, W only)
```

Example:

If DØ contains H'1B5A, then

```
XORW     .DØ, #H'FØFØ
```

changes the contents of DØ to H'EBAA.

Instruction Fields:

15	14	13	12	11	1Ø	9	8	7	6	5	4	3	2	1	Ø
Ø	Ø	Ø	Ø	1	Ø	1	Ø	size		mode			reg.		

+

	byte data (8 bits)

or

word data (16 bits)

or

long word data (32 bits, including previous word)

The size field contains the size of the operation.
If the size field is ØØ, it is byte.
If the size field is Ø1, it is word.
If the size field is 1Ø, it is long word.

The register and mode fields contain the address mode of the destination operand.

The data fields contain the data immediately following the instruction:
If the size field is ØØ, the the data is the low order byte of the immediate word (8 bits).
If the size field is Ø1, the the data is the entire

immediate word (16 bits).
If the size field is 1Ø, the the data is the next two immediate words (32 bits).

CHAPTER 9

DIRECTIVES

CHAPTER 9/ DIRECTIVES

Assembler directives are commands to the Assembler-16 (and not 68ØØØ machine instructions). A directive may produce code, affect the listing, or change the state of the assembly.

Directives may appear anywhere in the assembly source program, except for the END directive which must be the last statement of the program.

Each directive has a unique syntax.

Program Sections

Each source program processed by the Assembler-16 consists of one or more program sections (PSECTS). They are defined by the ASECT, DSECT and RSECT directives.

The ASECT directive defines the absolute section; there is a maximum of one. The RSECT defines a relocatable section; there may be zero or more relocatable sections in a program, but only one per assembly. The DSECT directive defines a dummy section; there may be zero or more dummy sections in a program. Note: no object is output for a DSECT.

A section of code is delineated by section directives. Code in the source program following a section directive belongs to the directive-defining section until another section directive is encountered.

Sections may be named or blank. Each named, relocated section is assumed to be independently relocatable from all other relocatable sections which are blank or named differently.

Only one blank (unnamed) relocatable section is recognized. RSECT directives with different names define different sections. The name of a section may not match the name of any other section of a different type.

Absolute (ASECT) section is used to generate non-relocatable code or data references. Addresses of data or code inside an absolute section cannot be altered by the linker or loader. It is recommended, therefore, that ASECT sections precede RSECT sections when coding.

Relocatable (RSECT) sections are used to produce object code which is relocatable by the linker or loader.

DSECT (dummy section) is used to generate absolute offsets. No object is output for a DSECT.

If neither an RSECT, DSECT, or ASECT statement begins an assembly, the blank (unnamed) RSECT is assumed.

ASECT
Absolute SECTion

[global label] ASECT

Opens an absolute segment and sets the current location counter to zero. Code assembled in the absolute segment may not be relocated by the linker or loader.

The label, if any, is defined at the current location counter after the segment is opened.

Example:

```
MAIN     ASECT
```

defines the beginning of an absolute segment with the name MAIN.

RSECT
Relocatable SECTion

[global label] RSECT

A relocatable segment is opened with the name defined by the label, if present, with the location counter initialized to zero.

Example:

```
SORT      RSECT
```

defines the beginning of a relocatable segment with the name SORT.

DSECT
Dummy SECTion

[global label] DSECT

Opens a dummy segment with the name defined by the label, if present, with the program counter initialized to zero.

Example:

```
EQUATES  DSECT
```

defines the beginning of a dummy segment with the name EQUATES.

ORG
ORiGinate program

ORG exp

Sets the current location counter to the value defined by the expression. The current program section is not changed.

If the current program section is absolute, the expression must be absolute.

If the current program section is relocatable, the expression must evaluate to relative to this section.

The expression must be pass-one defined. In an absolute section, exp is the address of the beginning of the ASECT. In a relocatable section, exp is the number of bytes between the RSECT and a preceding section (if there is one) or between Ø (if no preceding section).

Example:

```
    ORG     H'AØØØ
```

sets the current location counter to H'AØØØ.

RES
REServe

[label] RES[m] exp

where

m = B, W, L or null

Increments the current location counter by the number of bytes specified by the length indicator and the value of the expression. The length indicator, m, defaults to W. When m is W or L, the PC is set to an even number before the space is reserved by incrementing by 1 if PC is odd.

Defines the label at the current location counter (even boundary if m = W or L).

The expression must be a pass-one defined absolute value.

Example:

```
TABLE     RESW     1ØØ
```

reserves a block of memory which is 2ØØ bytes long, the first byte of which is known as TABLE.

EQU
EQUate

global label EQU[m] exp
or
global label EQU[m].register

where

m = B, W, L, U or null

Defines the label to take on the value of the expression or register in the operand field. If the length indicator, m, is specified, then the label takes on that length attribute. If the length indicator is absent, the label assumes the length attribute of the expression or the register symbol.

The expression must be pass-one defined.

Example:

```
SLASH     EQU       '/
```

equates the word SLASH with the code for "/".

END
END of program

[label] END [exp]

Defines the label, if present, at the current location counter. The END directive is the last valid statement in the assembly. An error message is output on every source line after this.

If an end-of-file on input is reached prior to an END statement, a warning message is generated, and an END record is assumed.

The optional expression operand defines the entry point address, to which control is transferred after loading.

Note: When several programs are linked the exp is taken from the last program included.

Example:

```
FINISH      END      ENTRY1
```

defines the end of the program to be here, at statement label FINISH, and defines ENTRY1 to be the entry point of the program.

TITLE
TITLE of page

```
TITLE  'string'
       "string"
```

Causes a new page on the listing output, and the character string is set in the subheader of each subsequent page of the listing.

The character string should not be packed. It should begin and end with matching quotes, either single or double.

The TITLE directive is the first line past the page header on the new page.

Example:

```
     TITLE          'DATA FORMAT ROUTINE'
```

causes a new page to be printed, and the heading "DATA FORMAT ROUTINE" to be printed at its top.

PAGE
new PAGE

PAGE

Causes a new page on the listing output. The PAGE directive is the first line past the page header on the new page.

Example:

```
        PAGE
```

causes a new page to be printed.

DATA
define DATA

[label] DATA[m] expl[,exp2[,...,expn]]

where

m = B, W, L or null

Computes the values associated with the operand expressions and stores them in consecutive memory locations. Each expression occupies a byte, word, or long word depending on the length indicator: B, W, or L respectively. If no length indicator is given, default is W. If W or L is specified, the PC is adjusted to an even boundary before generation of data. The optional label takes on the value of the current location counter after adjustment (if any).

The length attribute of the label is set to B, W, or L according to the length indicator.

If the value is too large for the specified field, it is truncated and a warning is generated.

Example:

```
POWER2      DATAB          1,2,4,8,16
```

fills 5 bytes beginning at location POWER2 with the numbers given.

RDATA
Repeat DATA

[label] RDATA[m] expl, exp2

where

m = B, W, or null

Computes the value associated with the operand expression exp2 and stores it in 'expl' consecutive memory locations. Each exp2 occupies a byte, word or long word depending on the length indicator (B, W, or L respectively). If no length indicator is given, the default is W. If W or L is specified, the PC is adjusted to an even boundary before generation of data. The optional label takes on the value of the current location after adjustment (if any).

The length attribute of the label is set to B, W, or L according to the length indicator.

If the value is too large for the specified field, it is truncated and a warning is generated.

Note: expl must be a pass-one defined absolute value.

Example:

```
TABLE2      RDATAL          1ØØ,Ø
```

defines a 4ØØ byte area of memory as TABLE2 and fills it with zeroes.

TEXT
TEXT string

```
[label] TEXT 'string'
             "string"
            P'string'
            P"string"
```

Stores the operand character string in memory in either unpacked or packed format. In unpacked format, the string operand begins with a quote (single or double).

Two consecutive quotes inside a character string represent one quote character as part of the string. Characters are stored one ASCII, zero-parity character per byte. In packed format, the characters are first encoded and then packed, modulo 4Ø, three characters per word, occupying an integral number of words with blank fill on the right as necessary. The packing algorithm encodes the characters as:

letter	f(letter)
A-Z	1-26
Ø-9	27-36
$	37
-	38
	39
all others	Ø

These encoded letters are packed three to a word by setting the word equal to:

```
((f(C1)*4Ø+f(C2))*4Ø)+f(C3)
```

where multiplication and addition employ unsigned 16-bit arithmetic.

If a label is present, it is defined at the current location counter, with a length attribute of B.

Null character strings are valid, no output being generated.

Example:

```
MSG1     TEXT      'INSERT NEW DISK'
```

defines a block of memory called MSGI and fills it with the string.

TEXTC
TEXT string with Count

```
[label]  TEXTC 'string'
               "string"
              P'string'
              P"string"
```

Stores the characters of the operand as in the TEXT directive, preceded by a count of the number of characters in the string.

If the string is packed, the count occupies one word. If it is unpacked, the count occupies one byte.

If a label is present, it is defined at the current location counter with a length attribute of B.

Null character strings are valid, a one-word count of Ø being generated.

Example:

```
LINEIN          TEXTC           'STRING LENGTH'
```

defines a block of memory call LINEIN and fills it with the string, the first element being the string length.

REF
REFerence external symbol

global label REFm

where

m = B, W, L or null

The required label is declared to be an external symbol with a length attribute as defined by the length indicator, m. If no length attribute is specified, the default is W. All references to label in code-generating instructions will be updated by the linker. This restricts the complexity of expressions in which a label may appear to that which is manageable by the linker.

The reference is assumed to be a relocatable address.

Example:

```
USER1     REFW
```

declares the word variable USER1 to be an external variable.

DEF
DEFine external symbol

DEF symbol1[,symbol2[...,symboln]]

The symbol(s) are declared to external symbols which are defined within this module. The symbols must be global labels, and may be either absolute or relocatable. Symbols must be DEFined in order to REFerence them from other modules.

FORM symbols and user-defined op code directives cannot be DEF'd. Symbols that are external (i.e., declared by REF) cannot be DEF'd in the same assembly.

Example:

```
        DEF       ENTRY1, ENTRY2
```

defines ENTRY1 and ENTRY2 to be external labels defined within the current.

COPY
COPY filename

COPY filespes

where filespec is any valid TRSDOS-16 file specification consisting of filename and extension (no password, drive number, or diskname).

Copies the source file specified by filename into the input stream after this COPY statement and before the next statement in the normal input stream.

There is a maximum of nine COPY statements in the assembly. No nesting of COPY statements is allowed.

Example:

```
        COPY      STDIO
```

copies the file STDIO into the current location, in the program.

FORM
FORMat definition of data

global label FORMm expl[,exp2[...,expn]]

where

m = B, W, L or null

The FORM directive enables the user to define bits of data. The global label is defined to be a FORM-symbol (mnemonic). A FORM-symbol (mnemonic) must be defined by the FORM directive before it is referenced in a program.

The operand expressions (exp) specify bit-field sizes. They must be absolute and pass one-defined, and the sum of the expression values must be 8, 16, or 32 as the length indicator, m, is: B, W, or L respectively. If no length attribute is specified, the default is W. Each expression defines the number of bits occupied by a field in the generated user instruction which employs this form. Up to 32 fields may be defined for m = L; up to 16 fields may be defined for m = W; up to 8 fields may be defined for m = 8.

The first expression, (exp) describes the high-order bit field of the generated object data; the second (if any) expression the next contiguous field; .... until m is filled.

Example:

```
X4X12    FORMW 4, 12
```

describes a word with two fields of 4 (high order) and 12 bits respectively. X4X12 can then be used as a mnemonic (Formal Symbol) to generate a word format as follows:

```
        LabelA X4X12 3 H'3ØØ
```

where the bit fields will be ØØ11 and ØØ11ØØØØØØØØ; the first field is filled with a decimal 3 and the second with a hex 3ØØ.

Extended Use of FORM-symbol (second level directive)

```
global label FORM-symbol expl[,exp2[...,expn]]
```

The global label (second level mnemonic) is defined as yet another code-generating Assembler instruction. The code generated by a second level mnemonic is byte, word, or long word (depending upon how the FORM-symbol was defined.)

Note: The FORM-symbol must have been previously defined in a FORM directive.

The number of expressions appearing as operands must match the number of bit fields defined in the FORM directive. Each expression may contain reference parameters (&1, &2 ...,&n).

When a second level mnemonic is used in source code its operands replace the reference parameters (&1, &2...,&n) of the second-level directive. The data generated are the bit fields defined by the new expressions list.

With the earlier Example:

```
X4X12  FORMW  4,12
```

you can create a second mnemonic:

```
SECLEV  X4X12  3, &1+H'2Ø
```

The global label (SECLEV) becomes the second level mnemonic, used as follows:

```
[label]    SECLEV H'3ØØ
```

where the operand H'3ØØ replaces the reference parameter &1. The resulting data (bit fields) would be: 3, H'3ØØ+H'2Ø or ØØ11 ØØ11ØØ1ØØØØØ.

Note: The sizes of bit fields (4, 12) were given in the FORM directive.

CHAPTER VI

PRIVILEGED INSTRUCTIONS

As was said in an earlier chapter, the MC68ØØØ microprocessor has two modes of operation: the user mode, and the supervisor mode. This chapter deals with the additional instructions generated by the Assembler-16. They are available in the supervisor mode and are called "privileged" instructions.

The TRSDOS-16 Operating System works in the supervisor mode. However, in order to protect itself it does not permit the user to enter this mode. Therefore, there is no way to enter the supervisor mode while operating under TRSDOS-16.

AND
AND status register

AND[l] .SR, #exp[.W or .L]   Operand length(l): B, W

Performs a logical AND on the immediate expression (#exp) and the status register (SR), and stores the result in the status register.

Example:

If the SR contains H'2Ø15, then

```
     ANDB     .SR, #H'F6
```

changes the contents of the SR to H'2Ø14.

Condition Codes:

X	N	Z	V	C
*	*	*	*	*

All of the flags are set according to the operation.

Instruction Field:

15	14	13	12	11	1Ø	9	8	7	6	5	4	3	2	1	Ø
Ø	Ø	Ø	Ø	Ø	Ø	1	Ø	size			mode		reg.		
word data (16 bits)															

The size field contains the size of the operation.
If the size field is ØØ, it is byte.
If the size field is Ø1, it is word.
If the size field is 1Ø, it is long.

The register and mode fields contain the destination (status register) 111 1ØØ.

The word data field contains #exp.

LD
LoaD status register

```
LD[l] Destination source        Operand length(l): W
LD[l] .SR,  #exp[.Wor .L]
      .SR, -@As
      .SR, @As+
      .SR, /exp[.W or .L]
      .SR, [exp]@As[(Ri)]
      .SR, exp[@PC[(Ri)]]
```

Loads the contents of the source (second) operand into the status register (SR). The source operand is a word. All bits of the status register are affected.

Example:

If the SR contains H'231Ø, then

```
LDW     .SR, #H'2Ø15
```

changes the contents of the SR to H'2Ø15.

Condition Codes:

X	N	Z	V	C
*	*	*	*	*

All flags are set according to the source operand.

Instruction Field:

15	14	13	12	11	1Ø	9	8	7	6	5	4	3	2	1	Ø
Ø	1	Ø	Ø	Ø	1	1	Ø	1	1	mode			reg.		

The register and mode fields contain the address of the source operand.

MOV
MOVe status register

MOV[l] Destination, Source
MOV[l] .SR, .Ds **Operand length(l): W**

Moves the contents of a data register to the status register.

Example:

If DØ contains H'2Ø15, then

```
     MOVW     .SR, .DØ
```

changes the contents of the SR to H'2Ø15.

Condition Codes:

X	N	Z	V	C
*	*	*	*	*

All of the flags are set according to the source operand.

Instruction Field:

15	14	13	12	11	1Ø	9	8	7	6	5	4	3	2	1	Ø
Ø	1	Ø	Ø	Ø	1	1	Ø	1	1	mode			reg.		

The <u>register</u> and <u>mode</u> <u>fields</u> contain the source data register.

MOV
MOVe user stack pointer

MOV[l] Destination, source
MOV[l].Ad, .USP **Operand length(l): L**
.USP, .As

Moves the contents of the user stack pointer (USP) to or from an address register.

Example:

If the USP contains H'ØØFF ØØØØ, then

```
MOVL     .AØ, .USP
```

changes the contents of AØ to H'ØØFF ØØØØ.

Condition Codes:

X	N	Z	V	C
-	-	-	-	-

None of the flags are affected.

Instruction Field:

15	14	13	12	11	1Ø	9	8	7	6	5	4	3	2	1	Ø
Ø	1	Ø	Ø	1	1	1	Ø	Ø	1	1	Ø	dr	reg.		

The direction field contains the direction of the move.
If the direction field contains Ø, the contents of the user stack pointer is moved from an address register.
If the direction field contains 1, the contents of the user stack pointer is moved to and address register.

The register field contains the address register number used in the operation.

OR
inclusive OR status register

OR[l] .SR, #exp[.W or .L] Operand length(l): B, W

Performs a logical OR on the immediate expression (#exp) and the status register (SR), and stores the result in the status register.

Example:

If the SR contains H'2Ø15, then

```
     ORW     .SR, #H'AØØØ
```

changes the contents of the SR to H'AØ15.

Condition Codes:

X	N	Z	V	C
*	*	*	*	*

All of the flags are set according to the operation.

Instruction Field:

15	14	13	12	11	1Ø	9	8	7	6	5	4	3	2	1	Ø
Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	size		mode			reg.		
word data (16 bits)															

The size field contains the size of the operation.
If the size field is ØØ, it is byte.
If the size field is Ø1, it is word.
If the size field is 1Ø, it is long.

The register and mode fields contain the destination status register. 111 1ØØ

The word data field is the expression.

RESET
RESET external devices

RESET **Operand length(1): unsized**

Resets all external devices. The processor state, other than the program counter, is unaffected and execution continues with the next instruction.

Example:

If the PC contains H'6ØØ4, then

```
    RESET
```

resets all external devices and increments the PC to H'6ØØ6.

Condition Codes:

X	N	Z	V	C
-	-	-	-	-

None of the flags are a affected.

Instruction Field:

15	14	13	12	11	1Ø	9	8	7	6	5	4	3	2	1	Ø
Ø	1	Ø	Ø	1	1	1	Ø	Ø	1	1	1	Ø	Ø	Ø	Ø

RETI
RETurn from Interrupt

RETI **Operand length(1): unsized**

Pulls the status register (SR) and the program counter (PC) from the system stack. The previous status register and program counter are lost. All bits in the status register are affected.

Example:

If the SP points to memory address H'6ØØØ which contains H'ØØ 15 ØØ ØØ FØ ØØ, then

```
RETI
```

changes the contents of the SR to H'ØØ15 and the PC to H'FØØØ.

Condition Codes:

X	N	Z	V	C
*	*	*	*	*

All of the flags are set according to the content of the word on the stack.

Instruction Field:

15	14	13	12	11	1Ø	9	8	7	6	5	4	3	2	1	Ø
Ø	1	Ø	Ø	1	1	1	Ø	Ø	1	1	1	Ø	Ø	1	1

WAIT
WAIT for interrupt

WAIT #exp[.W or .L] **Operand length(1): unsigned**

Moves the immediate operand (#exp) into the entire status register (SR). The program counter (PC) is advanced to the next instruction, and the processor stops until an interrupt, reset or trace occurs.

A trace exception will occur if the trace state is on when the WAIT is executed. If an interrupt request arrives with a priority higher than the current processor priority, an interrupt exception occurs. If the bit corresponding to the S-bit of #exp is off, execution of the instruction will cause the privilege violation. External reset will always initiate reset exception processing.

Example:

If the SR contains H'2Ø11, then

```
WAIT      #H'2Ø15
```

changes the contents of the SR to H'2Ø15 and stops all processing until an interrupt is received.

Condition Codes:

X	N	Z	V	C
*	*	*	*	*

All the flags are set according to #exp.

Instruction Field:

15	14	13	12	11	1Ø	9	8	7	6	5	4	3	2	1	Ø
Ø	1	Ø	Ø	1	1	1	Ø	Ø	1	1	1	Ø	Ø	1	Ø
#exp															

The #exp field contains data to be loaded in the status register.

XOR
eXclusive OR status register

XOR[l] .SR #exp[.W or .L] Operand length(l): B, W

Performs a logical eXclusive OR on the immediate expression (#exp) and the status register (SR), and stores the result in the status register.

Example:

IF the SR contains H'2Ø15, then

```
     XORB     .SR, #H'1F
```

changes the contents of the SR to H'2ØØA.

Condition Codes:

X	N	Z	V	C
*	*	*	*	*

All the flags are set according to the operation.

Instruction Field:

15	14	13	12	11	1Ø	9	8	7 6	5 4 3	2 1 Ø
Ø	Ø	Ø	Ø	1	Ø	1	Ø	size	mode	reg.
word data (16 bits)										

The size field contains the size of the operation.
If the size field is ØØ, it is byte.
If the size field is Ø1, it is word.
If the size field is 1Ø, it is long.

The register and mode fields contain the destination status register 111 1ØØ.

The word data field contains the #exp.

APPENDIX A/ Object Code Description

The object code produced by the assembler-16 serves as input to a linker routine before being stored in memory.

The object code defined here is designed to be compact and simple, yet capable of providing for future language extensions.

GENERAL STRUCTURE

The object code for a program is contained in a single file of sequential organization. The file consists of sequentially organized groups of data, called plexes, which may span record boundaries arbitrarily.

Each plex contains a byte count as its first character. This limits the size of a plex to 255 characters. The byte count includes the character containing the byte count. Each plex also contains the plex type as its second character. Thus, the minimum size for an object code plex is two bytes.

Symbols within the object code are of variable length. In conformance with the assembler-16 limit, a maximum of 45 characters for each symbol is observed. Whenever a symbol appears in the object code, the first byte is a count, and the subsequent bytes contain the symbol in unpacked format.

PLEX TYPES

Define Processor (Ø)

Byte(s)	Contents
Ø	Plex size = 11
1	Plex type = Ø
2	Version = Ø
3	Language Processor = Ø
4	Maximum Symbol Size = 45
5-1Ø	Date

The Define Processor (Plex Type Ø) group must be the first group of an object file.

The Version field refers to the version of the object code. Subsequent versions may make the prior versions obsolete and/or incompatible. The linker can use this information to support selected versions.

The Language Processor field contains a zero for the assembler-16 - other language processors will be assigned other numbers.

The Maximum Symbol Size in the object code is currently 45. The assembler-16 removes all blanks from symbols.

The Data refers to the date of object file creation. Date is ASCII in the form YYMMDD.

Define Program Section (1)

Byte	Contents
Ø	Plex Size
1	Plex Type = 1
2-3	Section Number
4-n	Section Name

This plex defines a section of code (Relocatable or Absolute) by name and by number. Subsequent reference to this section will be by number alone. The section number is unique for each differently named RSECT. An unnamed RSECT is given the number 1, and the ASECT is given the number Ø.

Section Name may contain a blank symbol of length 1 (to signify unnamed RSECT or ASECT); otherwise, it is the name given to the RSECT or ASECT assembly language program.

The effect of Define Program Section in the linker is to select the section as the current section and to set the location counter to zero.

Select Section (2)

Byte	Contents
Ø	Plex Size = 4
1	Plex Type = 2
2-3	Section Number

This plex selects a section to be the "current" section for subsequent object code groups. Only one section can be current at any point in the object code.

Another effect of the Select Section is to quiesce the previously current section. The program location counter of the quiesced section is not changed by the occurrence of this plex.

The location counter previously current when this section was last quiesced becomes the current location counter.

Select Section ORG (3)

Byte	Contents
Ø	Plex Size = 8
1	Plex Type = 3
2-3	Section Number
4-7	Displacement

This plex selects a section as current and sets the program location counter to a value relative to the beginning of the section.

The Displacement is a 32-bit number which defines the value of the new location counter for the section specified. The Displacement is relative to the start of the section.

Define Section Length (4)

Byte	Contents
Ø	Plex Size = 8
1	Plex Type = 4
2-3	Section Number
4-7	Section Length

This plex type defines the size of the referenced section (in bytes). Only one of these plex types per section is permitted, and there must be one for each relocatable section defined. This is not required for ASECT.

Define Symbol (5)

Byte	Contents
Ø	Plex Size
1	Plex Type = 5

2-3	Section Number
4-7	Value
8-n	Symbol
n+1...	More definition groups

This plex defines the value of a symbol.

The Section Number identifies the section relative to which the symbol is defined. If the definition is an absolute number, the Section Number should be zero.

The Value is 32 bits long and contains an offset from section start if Section type is relocatable, or an absolute number if Section type is absolute.

The (Section Number, Value, Symbol) group may be repeated within the plex to define arbitrarily many symbols.

Declare Symbol Reference (6)

Byte	Contents
Ø	Plex Size
1	Plex Type = 6
2-3	Symbol Number
4-n	Symbol
n+1...	More reference groups

This plex declares a symbol as an external-referenced name and assigns a number to it for subsequent usage in an object code expression.

The Symbol Number is a two-byte binary value, which is unique to this symbol among all symbols in this program file. This number is used in object code expressions to reference the symbol value.

As many (Symbol Number, Reference Size, Symbol) groups as necessary may follow the first such group.

Load Constant Data (7)

Byte	Contents
Ø	Plex Size
1	Plex Type = 7
2-n	Date

This plex loads data at the current location counter and increments the location counter by one for each data byte loaded.

Load Constant Repeat Data (8)

Byte	Contents
Ø	Plex Size
1	Plex Type = 8
2-5	Repeat Count
6-n	Repeated Constant

This group loads repetitious data values into memory at the current location counter. The value to be repeated may be any number (>Ø) of bytes as computed from the size of the plex. The pattern is repeated the number of times specified in the repeat count. This can be used to repeat bytes, words, longwords, text strings, etc.

Load Data with Reference (9)

Byte	Contents
Ø	Plex Size
1	Plex type = 9
2	Modifiable Field Start
3	Modifiable Field Size
4	Skeleton Data Size
5-n	Skeleton Data
(n+1)-m	Object Code Expression

This plex loads data at the current location counter after evaluating an object code expression, including the result as part of the data.

The data to be loaded may be a byte, word, or longword.

The Modifiable Field Start is the bit number (Ø-31) within the Skeleton data where the modifiable field begins. The Modifiable Field Size gives the number of bits in the modifiable field, contiguous from Modifiable Field Start.

The Skeleton Data Size is 1, 2, or 4 for Byte, Word, or Longword respectively. The Skeleton Data contains the data value to be loaded (1, 2, or 4 bytes).

The Object Code Expression consists of a byte count followed by Polish notation for an expression to be evaluated by the

linker, the result of which is stored in the modifiable field of the data word. The byte count includes itself.

The object code expression polish consists of operands and operators combined according to the following rules:

The polish is in the form:

```
        <operand>
    or  <binary operator><operand><operand>
    or  <unary operator><operand>
```

The <operand> may itself be the polish for an expression, or it may be:

- a binary value

- an address

- the value of an external symbol

If the <operand> is the polish for another expression, then the first byte must be an operator. The operators are, therefore, numbered from 128 onward to distinguish them from operands. The values for the defined operators are given below.

Operands may occupy one or more bytes, dependent upon the operand type. The first byte of each operand is the type byte. Subsequent bytes define the value associated with the operand (if any).

Operand Type	Byte Length	Meaning
Ø	5	Binary value
1	7	Address
2	3	External Symbol

Contents of extra bytes in operand are:

```
Binary value    :    4 bytes 32-bit value
Address         :    2 bytes section number,
                     4 bytes offset
External Symbol:     2 bytes symbol number.
```

Operator Values and Types

Operator	Value	Type
-	128	unary
-	129	binary
+	13Ø	binary
*	131	binary
/	132	binary
.AND.	133	binary
.XOR.	134	binary
.OR.	135	binary
.NOT.	136	unary
.SHL.	137	binary
.SHR.	138	binary

The expression is evaluated using 32-bit values and accumulators.

The result must "fit" into the Modifiable Field. Sign extension is permitted; if the resultant value X is in the range:

$$-2^{n-1}-1 < X < 2^{n-1}$$

where n = number of bits in modifiable field.

Then x is considered to "fit".

Declare Program Entry Point (1Ø)

Byte	Contents
Ø	Plex Size = 8
1	Plex Type = 1Ø
2-3	Section Number
4-7	Entry offset

This plex defines the entry point of the program, to which control is passed when loading is complete.

End of File (255)

Byte	Contents
Ø	Plex Size = 3

1	Plex Type = 255
2	Error Indicator

This plex terminates the program, closes all sections, and signals the end of the file.

The error indicator byte signifies what diagnostics were generated by the assembler-16.

Ø = clean assembly
4 = warnings generated
8 = errors generated

APPENDIX B / Memory Map

```
BEGINNING OF -->  ---------------------
  MEMORY          |                   |
                  |    TRSDOS-16      |
                  |                   |
                  ---------------------
                  |      DEBUG        |
                  |    IF LOADED      |
                  |                   |
                  ---------------------
                  |    RUNCOBOL       |
                  |    IF LOADED      |
                  |                   |
BASE ADDRESS -->  ---------------------
                  |                   |
                  |                   |
                  |                   |
                  |       USER        |
                  |      MEMORY       |
                  |                   |
                  |                   |
                  |                   |
BOUNDS ADDR   --> ---------------------
```

User memory begins at H'5ØØØ if the Debugger is not configured and at H'6ØØØ if the Debugger is configured.

APPENDIX C/ SAMPLE PROGRAMS

PROGRAM 1: Simple 16 Bit Binary Addition

This program adds the contents of one memory word (NUM1) to another (NUM2), and stores the result in a long word (SUM).

```
        RSECT
NUM1    DATAW   H'1ØFB
NUM2    DATAW   H'FF17
SUM     RESL    1
*
BEGIN   CLRL    .DØ              Clear DØ
        LDW     .DØ,/NUM1        Put the word at NUM1 into DØ
        ADDW    .DØ,/NUM2        Add the word at NUM2 to DØ
        STL     .DØ,/SUM         Store the long word at SUM
        END     BEGIN
```

After execution of this program, the number H'ØØØ1 1Ø12 is stored in the memory location associated with SUM.

PROGRAM 2: BCD Addition

This program adds two binary coded decimals (BCD's) stored in NUM1 and NUM2, and stores the result in a third memory location SUM. (Note: Remember that all BCD arithmetic is byte size only!)

```
        RSECT
NUM1    RESB    1
NUM2    RESB    1
SUM     RESB    1
*
BEGIN   CLRL    .DØ
        LDB     .DØ,/NUM1        Load the number at NUM1 into DØ
        LDB     .D1,/NUM2        Load the number at NUM2 into D1
        ADDD    .DØ,.D1          Add the two numbers
        STB     .DØ,/SUM         Store the resulting byte in SUM
        END     BEGIN
```

PROGRAM 3: Search a Table

This program searches a TABLE for an ITEM. The word size elements of TABLE are arranged in ascending order, with the first word being the number of elements in the TABLE. If the ITEM is found, its number in the TABLE is returned as ITEMNO. If no match is found, then ITEMNO is set to H'FF.

```
          RSECT
ITEM      RESW     1
TABLE     DATAW    H'5,H'9,H'15,H'35,H'4Ø,H'F3
ITEMNO    RESB     1
*
BEGIN     LDL      .AØ,#TABLE       Put start address of TABLE in AØ
          LDW      .DØ,@AØ+         Put number of elements in DØ
          ADDL     .AØ,.DØ          Point AØ 1 word past last elementof
          ADDL     .AØ,.DØ          of TABLE (twice since word=2 bytes)
          SUBW     .DØ,#1           Setup DØ for DB instruction
          LDB      .D2,#FF          Assume that no match will be made
          LDW      .D1,/ITEM        Put ITEM into DØ
LOOP      CMPW     .D1,-@AØ         Check element pointed to by AØ; if
          DBGE     .DØ,LOOP         DØ <> -1 and D1 < element, continue
          BNE      DONE             If not equal then branch to DONE
          MOVB     .D2,.DØ          Else move element number to D2
DONE      STB      .D2,/ITEMNO      Store element number in ITEMNO
          END
```

If ITEM equals H'15, then after execution of the program, ITEMNO contains H'Ø2; if ITEM equals H'36, then ITEMNO is set to H'FF.

PROGRAM 4: Convert ASCII to binary

This program converts an ASCII coded decimal to its binary value. The program has two basic steps, first, conversion of the ASCII coded decimal into a binary coded decimal, and second, conversion of that BCD into its binary equivalent. Step 1 is accomplished by subtracting #H'3Ø from the ASCII number (for example, the ASCII code for 7 is H'37, so H'37 - H'3Ø = H'Ø7). To accomplish step 2, we simply multiply the BCD by a power of 1Ø based on its position in the ASCII string.

```
          ASECT
          ORG      H'4ØØØ
ASCINT    RESL     1
HEXNUM    RESW     1
*
```

```
BEGIN     CLRL    .D2             Clear the sum register (D2)
          LDL     .DØ,#1          Set initial power of 1Ø to 1
          LDA     .AØ,ASCINT+4    Put address of 4 past ASCINT in AØ
LOOP      LDB     .D1,-@AØ        Get byte of ASCINT(start from left)
          SUBB    .D1,#H'3Ø       Convert to BCD
          MULU    .D1,.DØ         Multiply by power of 1Ø
          ADDL    .D2,.D1         Add product to sum
          MULU    .DØ,#1Ø         Raise DØ to next power of 1Ø
          CMP     .AØ,#ASCINT     Check to see if at leftmost byte
          BNE     LOOP            If not, then continue
          STW     .D2,/HEXNUM     Else store sum in HEXNUM
          END     BEGIN
```

If ASCINT contains "1245" (coded H'31 32 34 35), then at the end of the program, HEXNUM contains H'Ø4DD.

PROGRAM 4A: Program 4 as a subroutine

Program 4 represents a commonly used routine. For example, after inputting a number with the KBLINE SVC, you might need to perform an arithmetic operation with it. But to use it, you must first convert the ASCII code, which the SVC returns, to a binary number, which the computer uses.

In this example, the subroutine requires that a KBLINE SVC has just been executed successfully. The subroutine processes the number just as in Program 4, and returns the binary number to the calling program by putting it at the top of the user stack (A7).

(Note: The 68ØØØ limits multiplication to single word operands; however, to convert a large number, we need power of ten which takes up more than one word. To overcome this, we simply treat the multiplicand, i.e., the power of ten, as two separate words.

We then multiply the BCD by the low word, store it, and then shift the high word into the low word. The BCD is again multiplied by this word, but before we add it to the low-word-product we shift it back to the high word.

For simplicity's sake, we limit the input number to less than 9 characters, so that we keep the shifted high word product in one register. However, if you need to convert a longer number, then you may use more than one register to contain the operands and the product.

```
*Subroutine ASCII TO HEX
*
*Purpose
*    Convert an ASCII string to its binary numberic value.
*
*Entry Conditions
*    An ASCII string has been entered via the KBLINE SVC.
*
*Exit Conditions
*    The converted string is put onto the top of the stack.
*
*Limitations
*    The value of the ASCII string must lie between -9,999,999 and
* 99,999,999.  If the string is too large, or if it contains a non-
* numeric character (other than a leading minus sign), the subroutine
* returns H'8ØØØ ØØØØ to the stack.
*
ASCII TO HEX
     STML      .DØ,.D1,.D2,.D3,.D4,.D5,.A1,.A2,-@A7    *Save registers
     LDW       .D5,#H'1Ø               *Put shift value into D5
     LDW       .D4,12@AØ               *Get length of input string
     CMPW      .D4,H'Ø9                *Check for too big
     BGT       ERROR                   *If too big then branch
     SUBB      .D4,#H'Ø1               *Else continue; correct for DBC
     CLRL      .D3                     *Clear out sums register
     LDL       .A1,#POWERS             *A1 points to POWERS
     LDL       .A2,8@AØ                *A2 points to buffer from SVC call
     ADDW      .A2,12@AØ               *A2 points to 1 byte past buffer
     CMPW      14@AØ,#H'ØD             *Check last character for CR
     BNE       NEXT CHARACTER          *If not, then branch
     SUBL      .A2,#H'Ø1               *Else correct length
     SUBB      .D4,#H'Ø1               *Correct counter
*
NEXT CHARACTER
     LDB       .DØ,-@A2                *Take a byte of the string
     SUBB      .DØ,#H'3Ø               *Convert ASCII to BCD
     BN        NON NUMERIC             *If negative result, then branch
     CMPB      .DØ,#H'Ø9               *Compare with high bounds
     BGT       ERROR                   *If greater than 9, then branch
     LDL       .D1,@A1+                *Get current power of 1Ø
     MOVL      .D2,.D1                 *Copy D1 in D2
     MULU      .D1,.DØ                 *Multiply the BCD by a POWER
     ADDL      .D3,.D1                 *Put product in sums register
     SHRL      .D2,.D5                 *Move high word to low word
     BE        NO HIGH                 *If D2 = Ø, then branch
     MULU      .D2,.DØ                 *Multiply the BCD by high word
```

```
     SHLL      .D2,.D5                *Shift low word back to high word
     ADDL      .D3,.D2                *Add product to sum register
*
NO HIGH
     DBC       .D4,NEXT CHARACTER     *If more numbers, then branch
     BR        SUB DONE               *Else branch to SUB DONE
*
NON NUMERIC
     ADDB      .DØ,#H'3Ø              *Convert BCD back to ASCII
     CMPB      .DØ,#H'2D              *Check for negative sign code
     BNE       ERROR                  *If not right code, then branch
     CMPB      .DØ,#H'2D              *Make sure it's the lefmost byte
     BNE       ERROR                  *If not, then branch
     NEGL      .D3                    *Else take negative of sum register
     BR        SUB DONE               *Branch to SUB DONE
*
ERROR
     LDL       .D3,#H'8ØØØØØØ         *Error code is H'8ØØØ ØØØØ
*
SUB DONE
     STL       .D3,H'24@A7            *Return result to stack
     LDML      .DØ,.D1,.D2,.D3,.D4,.D5,.A1,.A2,@A7+    *Restore reg's
     RET                              *Return to calling routine
*
POWERS
     DATAL     1,1Ø,1ØØ,1ØØØ,1ØØØØ,1ØØØØØ,1ØØØØØØ
     DATAL     1ØØØØØØØ,1ØØØØØØØØ,1ØØØØØØØØØ
```

Note that the first thing we did was save the contents of the registers which the subroutine altered. The STM and LDM instructions, in combination with pre-decrement and post-increment indirect addresssing work well for this since the STW instruction stores from the lowest address register to the highest data register, while the LDM instruction loads in just the opposite order.

PROGRAM 5: Converting Hex to ASCII Integer

The last program described how to convert an ASCII integer, presumably entered from the keyboard, into its binary (or hex) equivalent. It is also important to be able to do the opposite--convert the binary to ASCII integer, so that it can be sent to a display.

To accomplish this, we basically need to run Program 4 in reverse. That is, we can divide the binary number by powers of ten to give the BCD value of each decimal digit. We then

convert the BCD to ASCII by adding H'3Ø, and store the values in a string.

However, as in Program 4A, we run into the problem of size. The 68ØØØ can accept division of a long word by a single word, resulting in another single word. However, we may have some numbers which would result in a quotient longer than one word--this would give an overflow error.

Because of this, we must perform the division by subtracting a power of ten, repeatedly (summing the number of times we subtract) until the difference is less than that power of ten. We then perform the same steps with lower powers of ten until the power of ten equals one. The number of the subtractions per power gives the BCD for that power. It is a simple matter, then, to convert the BCD string into an ASCII string by adding H'3Ø to the BCD.

```
*Subroutine HEX TO ASCII
*
*Purpose
*    Convert a hexadecimal (binary) number into its ASCII-coded
* decimal equivalent.
*
*Entry conditions
*    The value of the hex number is pushed to the stack, on top of the
* destination address of the resulting ASCII string.
*
*Exit conditions
*    The ASCII string is stored in the memory address specified by the
* calling program.
*
*Limitations
*    The hex number must evaluate to a decimal value between
* +/- 1Ø**9.
*
HEX TO INT
     STML      .DØ,.D1,.D2,.AØ,.A1,-@A7   *Save registers
     LDA       .AØ,POWERS2              *Put address of POWERS2 in AØ
     LDL       .A1,H'18@A7              *Get address of ASCII INT off stack
     LDL       .DØ,H'1C@A7              *Get value of hex number off stack
     MOVB      @A1+,#H'2Ø               *Store blank for first character
     TESTL     .DØ                      *Check hex number for sign
     BP        NEXT POWER               *If positive then blank was correct
     SUBW      .A1,#H'Ø1                *Else move back to first character
     MOVB      @A1+,#H'2D               *Put a negative sign there
     NEGL      .DØ                      *Change the hex number to positive
*
```

```
NEXT POWER
     LDB        .D2,#H'3Ø              *Set D2 to ASCII zero
     LDL        .D1,@AØ+               *Get next power
*
MORE SUBTRACTION
     CMPL       .DØ,.D1                *Check for done with that power
     BLT        NEXT DIGIT             *If done then branch out of loop
     SUBL       .DØ,.D1                *Else subtract power from hex num
     ADDB       .D2,#H'Ø1              *and increment counter
     BR         MORE SUBTRACTION       *Continue subtraction
*
NEXT DIGIT
     STB        .D2,@A1+               *Store counter in ASCII INT string
     CMPB       .D1,#H'Ø1              *Check for last power
     BNE        NEXT POWER             *If not then do another power
     LDML       .DØ,.D1,.D2,.AØ,.A1,@A7+      *Else restore registers
     RET
*
POWERS2
     DATAL      1ØØØØØØØØØ,1ØØØØØØØØ,1ØØØØØØØ,1ØØØØØØ,1ØØØØØ
     DATAL      1ØØØØ,1ØØØ,1ØØ,1Ø,1
```

Compare the Program 5 with Program 4A. Note that when we loaded the address of POWERS in Program 4A, we used a load address register immediate (LDL) instruction, while in Program 5 we used a load effective address (LDA) instruction. Either is correct.

Also note that for Program 4A we ordered POWERS in ascending order and accessed the elements starting from the front, while in Program 5 we ordered POWERS2 in descending order and accessed the elements from back to front. Either is just as convienient since the 68ØØØ has auto increment (@An+) and decrement (-@An).

PROGRAM 6:Calling a subroutine

Programs 4A and 5 were presented as subroutines. It was assumed that some other program supplied the subroutine the necessary parameters (like the address of ASCII INT, etc.). Here is a sample showing what a main program which calls these subroutines might look like:

```
*Data block
*
     RSECT
HEXNUM
```

```
     RESL        1
SVC BLOCK
     RDATA       32,Ø
BUFFER
     RDATA       8Ø,Ø
ASCII INT
     RESB        12
     .
     .
     .
*
*Program block
*
     .
     .
     .
     LDA         .AØ,SVC BLOCK          *Set up KBLINE SVC
     MOVW        @AØ,#H'Ø5
     MOVW        6@AØ,#H'Ø9
     MOVL        8@AØ,#BUFFER
     BRK         #Ø
     TESTW       2@AØ                   *Check for SVC error
     BNE         ERROR ROUTINE          *On error branch to ERROR ROUTINE
     SUBL        .A7,#H'Ø4              *Make space on stack for hex num
     CALL        ASCII TO HEX           *Call up ASCII TO HEX subroutine
     CMPL        @A7,#H'8ØØØØØØØ        *Check for subroutine error
     BE          ERROR ROUTINE          *On error branch to ERROR ROUTINE
     STL         @A7+,/HEXNUM           *Pop hex number into HEXNUM
     .
     .
     .
     STL         /HEXNUM,-@A7           *Push hex number to stack
     PUSHA       ASCII INT              *Put address of ASCII INT on stack
     CALL        HEX TO INT             *Call up HEX TO INT subroutine
     MOVW        @AØ,#H'Ø9              *Set up VDLINE SVC
     MOVW        6@AØ,#H'ØB
     MOVW        8@AØ,#H'ØD
     MOVL        1Ø@AØ,#ASCII INT
     BRK         #Ø
     .
     .
     .
```

APPENDIX D/ The Configuration Command File

Whenever TRSDOS-16 starts up or is reset, it looks for a file named CONFIG16/SYS. This "configuration command file" tells TRSDOS-16 to link in certain extra operating system programs.

CONFIG16/SYS should be present on the primary disk device (drive Ø or drive 4). It contains these directives:

```
INCLUDE RUNCOBOL
INCLUDE DEBUG
END
```

which tell TRSDOS-16 to link in the RUNCOBOL program and the DEBUG program.

You may create your own CONFIG16/SYS file, or modify the existing one to meet your needs, by using EDIT16.

SAVING THE EXISTING CONFIG16/SYS FILE

Before creating a new CONFIG16/SYS file, you will probably want to save the existing one by renaming it.

For example:

```
RENAME CONFIG16/SYS:Ø TO DEBCOB/CFG:Ø
```

renames the default configuration file. (The new filename tells you it includes both DEBUG and RUNCOBOL modules.

After renaming the existing CONFIG16/SYS file, you can create a new one.

Since you "saved" the existing file, you can use it again. To do this, rename the present CONFIG16/SYS file (if you want to save it) and then rename DEBCOB/CFG back to CONFIG16/SYS:

```
RENAME DEBCOB/CFG:Ø TO CONFIG16/SYS:Ø
```

TO EDIT OR CREATE CONFIG16/SYS

Use EDIT16 to edit or create a CONFIG16 command file.

1. Type:

   EDIT16 <ENTER>

   and the Editor's Command mode prompt will be displayed:

   C?...........................

2. To insert commands into the command file, you must get in the Insert mode, type:

   IN <ENTER>

   The Editor will display the I? prompt, indicating that you are in the Insert mode.

3. You are now ready to insert the names of the programs you want linked to TRSDOS-16.

   Comments may be used. They are indicated by an asterisk (*) in the first column.

   The key word **INCLUDE** tells TRSDOS-16 the name of the program. The syntax for the INCLUDE statement is:

   INCLUDE filename

   The default extension for filespec is /SYS; it is optional. Drive numbers, disk ID and Passwords are not permitted.

   Programs are loaded sequentially in memory in the order they are encountered in the CONFIG16/SYS file. The maximum number of programs that may be INCLUDEd is 15.

   The programs must be resident on the primary boot device (Drive Ø or Drive 4).

   The list is concluded with an **END** statement.

   For example:

   ```
   * This is the Configuration File for DEBUG
   INCLUDE DEBUG
   ```

END

tells TRSDOS-16 to link only the DEBUG program. The first line is a comment and is not executed by TRSDOS-16

4. When you are finished inserting, type:

   ! <ENTER>

   to exit the Insert Mode.

5. Save the file with the following command:

   SA CONFIG16/SYS <ENTER>

6. You now have a new CONFIG16 command file that TRSDOS-16 will use when it powers up or resets.

CONFIGURATOR ERROR MESSAGES

When the Configurator lists a line generating an error, it prints an error message directly underneath the line number. Preceding the message, it inserts three asterisks.

In cases of certain syntax or file I/O errors, the Configurator also marks, with a dollar sign ($), where in the line the error occurred.

For example:

```
Ø11 INCLUBE RUNCOBOL
    $
*** Illegal Command
```

shows a syntax error in the spelling of INCLUDE.

There are three catagories of Configurator error messages:

A. Configuration Control File Errors
B. Configuration Command Errors
C. Completion Errors

A. Congifuration Control File Errors

These errors are FATAL. If one of these errors occur, the Configurator could not properly execute the CONFIG16/SYS file. TRSDOS-16 will still be displayed but certain defaults will have occured:

1. No programs have been INCLUDED
2. DEBUG is kept resident (if available)
3. Any memory not occupied by DEBUG and the resident Operating System is available to the user.

Use EDIT16 to correct the error (or create a new configuration file) and reset the system.

Can't Open CONFIG16/SYS: TRSDOS Error Code = nnn

Look up TRSDOS-16 Error Code **nnn** in Appendix B and take appropriate action.

Can't configure system: File CONFIG16/SYS not proper format

The CONFIG16/SYS file is not a VLR type file.

Can't configure system: File CONFIG16/SYS not found

TRSDOS-16 could not find the CONFIG16/SYS file.

I/O Error on File CONFIG16/SYS: TRSDOS Error Code = nnn

Look up TRSDOS-16 Error Code **nnn** in Appendix B and take appropriate action.

B. Configuration Command Errors

These errors occur when a command cannot be processed by the Configurator. If one of these error occurs, the Configurator will continue to process the command lines. However, the desired result of the configuration file may not have been accomplished. For example, an INCLUDE file may have been left out.

Can't INCLUDE program: TRSDOS Error Code = nnn

The Configurator cannot load the program because of an I/O error. Look up the TRSDOS-16 Error Code in Appendix B.

Can't INCLUDE program: Out of Memory

More resident programs were requested than will fit into user memory.

Can't INCLUDE program: Program already configured

This error occurs any time a program is included twice.

Too many INCLUDED programs: this request ignored

This error occurs if more than 15 programs are included. The command line that is flagged is ignored (treated as a comment).

C. Completion Error

*** CONFIGURATION ABORTED ***

This message appears when the configurator could not finish processing the CONFIG16/SYS file because of an I/O error.

ABOUT THE CONFIGURATOR

The Configurator is invoked whenever the 68ØØØ processor is initialized. It performs several important functions:

- It determines whether the machine debugger is required. If not, it is eliminated from memory. This gives you an extra 4K of memory.
- It initializes traps and interrupts. This eliminates the need to keep extra code resident in memory.

- It loads in resident programs as specified in the CONFIG16/SYS file.
- It reads the AUTO file and passes it to TRSDOS-16 for execution.

The Configurator is linked in at the end of user memory and occupies 4K of memory. Upon system initialization, it moves itself to the top of physical memory. This is because the resident programs will be loaded at low address, overlaying the original configurator.

Next the Configurator begins to load the resident programs requested in the CONFIG16/SYS file (i.e. DEBUG and RUNCOBOL). It loads these programs sequentially starting at the beginning of user memory and up to the beginning of where the Configurator has relocated itself. This guarantees that after loading is complete, the user has at least 4K of memory available (the size of the Configurator).

After configuration is complete, the Configurator is no longer neccessary and is overwritten.

APPENDIX E/ Additional 68ØØØ Instructions

The MC68ØØØ microprocessor supports three opcodes in addition to those supported by the ASSEMBLER-16. These instrucions are BRanch Always (BRA), Branch to SubRoutine (BSR), and Test a Bit and Set (BSET).

BRA
BRA is very similar to BR exp(@PC) in that it allows you to transfer control of your program unconditional to a PC relative address. The major difference between the two instructions is that the BRA instruction can use less storage space when using an eight-bit displacement because it puts the displacement value in the same word as the instruction code, thus saving two bytes of storage spaces.

The BRA instruction may also be used for sixteen-bit displacements; however, the displacement is stored in an extension word following the instruction field. Like the BR instruction, the BRA instruction affects none of the conditions codes.

The instruction field for BRA is:

15	14	13	12	11	1Ø	9	8	7	6	5	4	3	2	1	Ø
Ø	1	1	Ø	Ø	Ø	Ø	Ø	8-bit displacement							
16-bit displacement															

For an eight-bit displacement, the second word (containing the sixteen-bit displacement) is not necessary. For branches using the sixteen-bit displacement, the eight-bit displacement field is filled filled with zeroes.

BSR
BSR is similar to the CALL exp(@PC) instruction, in that it allows you to call a subroutine whose start address can be given as a displacement relative to the program counter. As with CALL, the computer first pushes the address of the instruction following the BSR onto the stack. Also like CALL, BSR changes none of condition codes.

The major difference between CALL and BSR is that the BSR instruction may only be one word long when an eight-bit displacement is specified. For a sixteen-bit displacement, two words are used, the second word holding the displacement value.

The instruction field for BSR is:

15	14	13	12	11	1Ø	9	8	7	6	5	4	3	2	1	Ø
Ø	1	1	Ø	Ø	Ø	Ø	Ø	8-bit displacement							
16-bit displacement															

For an eight-bit displacement, the second word (containing the sixteen-bit displacement) is not necessary. For subroutine calls using the sixteen-bit displacement, the eight-bit displacement field is filled with zeroes.

BSET

BSET is similar to the other MC68ØØØ bit-test instructions. It tests a specific bit of the operand, records the result in the zero bit of the condition codes, and then sets the specified bit. You may specifiy the bit-number (i.e., the number of the bit to be tested) either from a data register or as an immediate value. The instruction field for the former is:

15	14	13	12	11	1Ø	9	8	7	6	5	4	3	2	1	Ø
Ø	Ø	Ø	Ø	Rs			1	1	1	Reg			Mode		

where Rs is a data register containing the bit number. The instruction field for the immediate version is:

15	14	13	12	11	1Ø	9	8	7	6	5	4	3	2	1	Ø
Ø	Ø	Ø	Ø	1	Ø	Ø	Ø	1	1	Reg			Mode		
Bit number															

The condition codes for BSET are:

X	N	Z	V	C
-	-	*	-	-

N Not affected
Z Set if the bit tested is zero, cleared otherwise
V Not affected
C Not affected
X Not affected

INDEX

RADIO SHACK A DIVISION OF TANDY CORPORATION

U.S.A.: FORT WORTH, TEXAS 76102
CANADA: BARRIE, ONTARIO L4M 4W5

TANDY CORPORATION

AUSTRALIA	BELGIUM	U. K.
280-316 VICTORIA ROAD RYDALMERE, N.S.W. 2116	PARC INDUSTRIEL DE NANINNE 5140 NANINNE	BILSTON ROAD WEDNESBURY WEST MIDLANDS WS10 7JN

8749339-482-SP

PRINTED IN U.S.A.